МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ОДЕСЬКИЙ ДЕРЖАВНИЙ ЕКОЛОГІЧНИЙ УНІВЕРСИТЕТ

Факультет комп'ютерних наук,
управління та адміністрування
Кафедра інформаційних технологій

Кваліфікаційна робота бакалавра

на тему: Розробка компонентів компілятора мови функціонального програмування Scheme

Виконав студент групи КН-5
спеціальність 122 комп'ютерні науки
Гуляк Роман Миколайович

Керівник д.т.н., професор
Зайцев Дмитро Анатолійович

Консультант _____
_____

Рецензент д.т.н., професор
Мещеряков Володимир Іванович

Одеса 2021

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

ANSI – American National Standards Institute – Американський інститут національних стандартів

AST – abstract syntax tree – абстрактне дерево синтаксису

MIT – Массачусетський технологічний інститут

x86-64 – архітектура системи команд

XML – Extensible Markup Language – розширювана мова розмітки

Desugaring – децукоризація – фаза компіляції

F# – мова програмування

Free variables – вільні змінні – змінні із зовнішнього оточування

M-expression – m-вираз – мета-вираз

Scheme – мова програмування

S-expression – s-вираз – символічний вираз

Syntactic sugar – синтаксичний цукор – конструкції мови програмування

ВСТУП

У цій роботі описана реалізація компонентів компілятора для мови Scheme. Це функціональна мова програмування, яка належить до діалектів мови Lisp. Компілятор виробляє код мовою асемблеру, потім виконує збірку виконуваного файлу.

Метою розробки даного компілятора було поглиблення знань в області формальних мов та застосування їх на практиці. У наш час практично кожен рік виникають нові мови програмування. Також постійно удосконалюються техніки реалізації компіляторів та інтерпретаторів. Даний проект призначався у якості «полігону для випробувань» таких технік. Перевага мови Scheme для таких цілей полягає в тому, що хоч вона є компактною мовою, але достатньо могутньою для написання серйозних програм.

Для реалізації компілятору використовувалась мова функціонального програмування F# [3]. Крім загального опису функціонування побудованого компілятору, в роботі детально описана реалізація двох компонентів: лексичного і синтаксичного аналізаторів. Кінцевою метою цих компонентів є побудова внутрішнього представлення вихідного коду програми для подальшого аналізу, оптимізації та перекладу в мову асемблера.

В роботі також показані приклади, завдяки яким читач може ближче ознайомитися із синтаксисом та особливостями Scheme, а також з результатами функціонування зазначених компонентів компілятора.

1 ІСТОРІЯ ТА ОСОБЛИВОСТІ МОВИ SCHEME

Scheme – це мова програмування загального призначення. Це мова високого рівня, що підтримує операції зі структурованими даними, такими як рядки, списки і вектори, а також операції з більш традиційними даними, такими як числа і символи.

Хоча Scheme часто пов'язують з символічними додатками, його багатий набір типів даних і гнучкі структури управління роблять його дійсно універсальною мовою. Scheme використовувався для написання текстових редакторів, оптимізаційних компіляторів, операційних систем, графічних пакетів, експертних систем, числових додатків, пакетів фінансового аналізу, систем віртуальної реальності і практично всіх інших типів додатків, які тільки можна уявити.

1.1 Історія створення

Scheme є діалектом мови Lisp. Крім Lisp на Scheme також вплинув мову ALGOL. Lisp надав свою загальну семантику і синтаксис, а ALGOL надав свою лексичну область видимості та блокову структуру.

Lisp був винайдений Джоном Маккарті в 1958 році, коли він працював в Массачусетському технологічному інституті (Massachusetts Institute of Technology, MIT). У своїй статті «Рекурсивні функції символьних виразів і їх машинне обчислення» [6] він показав, що за допомогою декількох простих операторів і позначень функцій можна побудувати повну по Тюрінгу алгоритмічну мову. Lisp був створений для вирішення задач обробки символьної інформації. Основна структура даних мови – список, звідси і назва мови: Lisp – List Processing.

Відмінною особливістю синтаксису Lisp та її діалектів є використання s-виразів (symbolic expression). Спочатку s-вирази використовувалися як проміжне представлення того, що Маккарті називав m-виразами (meta expression). Наприклад, m-вираз

```
head [cons [A, B]]
```

є еквівалентним s-виразу:

```
(head (cons A B))
```

Однак s-вирази виявилися популярними, а багато спроб реалізувати m-вирази не увінчалися успіхом.

Перший інтерпретатор Lisp був написаний Стівом Расселом на IBM 704, який прочитав статтю Маккарті і закодував описану їм функцію eval в машинному коді.

Перший повний компілятор Lisp, написаний на мові Lisp, був реалізований в 1962 році Тімом Хартом і Майком Левіним з Массачусетського технологічного інституту. Цей компілятор представив модель інкрементальної компіляції, в якій і скомпільовані й інтерпретовані функції можуть вільно змішуватися.

Мова Scheme була створена Гаєм Стилом і Джеральдом Сассменом в 1975 р. як спроба зрозуміти модель акторів Карла Хьюїтта. Першим назвою мови був "Schemer". Однак у зв'язку з обмеженнями довжини імен файлів операційної системи ITS, яка використовувалася творцями мови, назва була скорочена до "Scheme".

У 1991 р. для Scheme був виданий ANSI (American National Standards Institute) стандарт [7]. Після цього співтовариство створило серію менш форма-

льних звітів, "Revised reports", які розширювали ANSI стандарт. Зараз вони є де-факто стандартами Scheme.

Послідовність зазначених подій показана у табл. 1.1.

Таблиця 1.1 – Хронологія розвитку Scheme

Дата	Подія
1958	Створення Lisp
1962	Перший компілятор Lisp
1975	Створення Scheme
1991	ANSI стандартизація
1998	Revised5 report
2007	Revised6 report
2013	Revised7 report

1.2 Особливості мови Scheme

Scheme – досить проста мова для вивчення, оскільки в основі її є невелике ядро синтаксичних форм, з яких побудовані всі інші форми. Ці основні форми, набір розширених синтаксичних форм, отриманих з них, та бібліотека первісних процедур складають повну мову Scheme. Інтерпретатор або компілятор для схеми може бути досить малим і потенційно швидким і дуже надійним. Розширені синтаксичні форми та багато первісних процедур можуть бути визначені на самій мові Scheme, спрощуючи реалізацію та підвищення надійності.

Програми Scheme мають спільне зовнішнє представлення зі структурами даних Scheme. Як результат, будь-яка програма Scheme має природне та очевидне внутрішнє представлення як об'єкт Scheme. Наприклад, змінні та синтаксичні ключові слова відповідають символам, тоді як структуровані синтаксичні форми відповідають спискам.

Це представлення є основою для синтаксичних розширень, що надає змогу визначати у Scheme нові синтаксичні форми за допомогою існуючих синтаксичних форм і процедур. Це також спрощує реалізацію інтерпретаторів, компіляторів та інших інструментів трансформації програм.

В більшості мов визначення процедури є просто асоціацією імені з блоком коду. В цих мовах процедури можуть бути визначені лише на верхньому рівні. У Scheme визначення процедури може з'явитися в іншому блоці або процедурі, і процедура може бути викликана в будь-який час після цього визначення, навіть якщо блок, який закриває його, закінчив своє виконання. Для підтримки лексичного обстеження, процедура несе лексичний контекст (environment, навколишнє середовище) разом із його кодом.

Крім того, процедури Scheme можуть не мати ім’я. Такі процедури називаються анонімними. Процедури є об'єктами даних першого класу, такі як рядки або цифри, а змінні пов'язані з процедурами так само, як вони пов'язані з іншими об'єктами.

Так само як і в більшості інших мов, процедури у Scheme можуть бути рекурсивними. Тобто будь-яка процедура може викликати сама себе безпосередньо або опосередковано. Багато алгоритмів виражаються найбільш елегантно або ефективно саме через рекурсію. Спеціальний випадок рекурсії, що називається хвостова рекурсія, використовується для вираження ітерації або циклу. Хвостова рекурсія виникає, коли одна процедура безпосередньо повертає результат викликання іншої процедури. У Scheme хвостова рекурсія реалізована за допомогою стрибків (goto), і це дозволяє уникнути звичайні витрати на рекурсію.

Scheme підтримує визначення довільних контрольних структур завдяки продовженням (continuations). Продовження – це процедура, яка втілює решту програми за певною точкою в програмі. Продовження може бути отримано в

будь-який час під час виконання програми. Як і з іншими процедурами, продовженням є об’єктом першого класу і може бути викликаний у будь-який час після його створення. Всякий раз, коли воно викликано, програма негайно продовжується з тієї точки, де було отримано продовження. Продовження дозволяють реалізацію складних механізмів контролю, включаючи багатопоточність та корутини.

Відмінною особливістю Scheme і інших діалектів Lisp є наявність символів (symbols). Вони схожі на рядки, але відрізняються від рядків тим, що тільки об'єкт символу може мати дану послідовність знаків. Символи використовуються в програмах Scheme для різних цілей як символьні імена. Важливою характеристикою символів є те, що порівняння символів набагато більш ефективно, ніж порівняння рядків [1]. При порівнянні символів не потрібно перебирати всю послідовність знаків, досить порівняти адреси символів. Тому, символи можна використовувати в якості повідомлень для передачі між функціями, як мітки для структур або як імена для об'єктів, що зберігаються в асоціативних списках [4].

2 КОРОТКИЙ ОГЛЯД СИНТАКСИСУ МОВИ SCHEME

Scheme – це мова програмування високого рівня та загального призначення. Scheme підтримує операції зі структурованими даними, такими як рядки, списки та вектори, а також операції з більш традиційними даними, такими як числа.

Програми на Scheme складаються з ключових слів, змінних, структурованих форм, чисел, знаків, рядків, символів, спискових констант, пробілів і коментарів [1].

Ключові слова, змінні і символи разом називаються ідентифікаторами. Ідентифікатори повинні бути розділені пробілами, дужками, подвійними лапками (") або символом крапкою з комою (;). Мова Scheme надає набагато більший вибір ідентифікаторів, ніж інші мови. В ідентифікаторах можна використовувати букви, числа, спеціальні символи. Приклади ідентифікаторів: `hi`, `Hello-World`, `n`, `x34`, `?\$&\*!!!`.

Структуровані форми і спискові константи поміщаються в круглих дужках, наприклад `(a b c)`, `( )`. Замість круглих дужок можна використовувати квадратні дужки `[ ]`, напр. `(* [- x 2] [+ y 3])`. Вони покращують вигляд складних виразів.

Булеві значення, що представляють істину та хибність, записуються як `#t` та `#f`.

Вектори записуються як списки, за винятком того, що вони починаються з символів `#(`, напр. `#(this is a vector of symbols)`.

Рядкові константу записуються між двома подвійними лапками, наприклад, `"I am a string"`.

Вирази в Scheme можуть займати декілька рядків і для них не потребується явного роздільника. Число символів пробілу (пробіли і символи нового рядка) між виразами не є значними, як наприклад в мові Python.

Тому програми форматуються відступами для підкреслення структури коду за смаком автора. Коментарі можуть з’являтися на любому рядку програми Scheme, між точкою з комою (`;`) і кінцем рядку.

3 F# ЯК МОВА РЕАЛІЗАЦІЇ КОМПІЛЯТОРУ

F# – це функціональна мова програмування, з сімейства мов .Net framework. Крім функціональної парадигми, F# також підтримує імперативний і об'єктно-орієнтований стиль. Мова була розроблена Доном Саймом в Microsoft Research в Кембриджі у 2005 році.

Мова F# має ряд особливостей, які роблять його особливо привабливим при розробці компілятора. це:

- підтримка функціонального стилю програмування;
- розмічені об'єднання;
- зіставлення з шаблонами;
- автоматичне виведення типів.

3.1 Функціональне програмування

У табл. 3.1 наведені основні відмінності функціонального програмування від традиційного імперативного.

Таблиця 3.1 – Порівняння функціонального та імперативного програмування

Функціональне програмування	Імперативне програмування
Використання рекурсії	Використання циклів
За замовчуванням, дані є незмінними	Дані є змінними, присутні глобальні дані
Концентрується на тому, *який результат* потрібно отримати	Концентрується на тому, *як отримати* бажаний результат

Функціональне програмування – це стиль програмування, в якому основний акцент поставлений на використанні функцій і незмінних структур даних. Відмінні риси функціонального програмування:

- використання виразів замість інструкцій (таких як присвоювання);
- використання незмінних значень (immutable values) замість змінних (variables);
- більша частина функцій є чистими.

Чисті функції (pure functions) – це функції, які обчислюють один і той же результат для однакових аргументів і не мають побічних ефектів. Прикладами побічних ефектів є зміна глобальних змінних або внесення змін в базу даних.

Перевагою використання чистих функцій є спрощення їх розуміння і налагодження. Оскільки функція не залежить від оточення, її можна налагоджувати незалежно від решти системи. Також чисті функції надають можливості оптимізації, наприклад, кешування. Це обумовлено тим, що якщо функція повертає однаковий результат для тих же аргументів, то обчислення можна робити тільки для тих аргументів, які ще не зустрічалися.

За замовчуванням, у F# змінні та структури даних є незмінними, тобто після ініціалізації змінна не змінює свого значення. Аналогом такої поведінки, наприклад, в Java було б додавання ключового слова `final` до кожної змінної. Використання незмінних даних сприяє зниженню помилок пов'язаних з даними, які часто змінюються. Однак, і в F# можна використовувати змінювані змінні за допомогою ключового слова `mutable`.

3.2 Розмічені об’єднання

Розмічене об'єднання (discriminated union) – це тип, який визначається користувачем. Значення цього типу можуть бути одним з іменованих варіантів, при чому можливо, що ці варіанти мають різну кількість і типи полів. Розмічені об'єднання також підтримують рекурсивні визначення, дозволяючи з легкістю визначати такі структури як дерева та інші рекурсивні дані.

Наприклад, можливо визначити бінарне дерево пошуку таким чином:

```
type BST =
    | Empty
    | Node of string * BST * BST
```

Кожне значення зазначеного типу може бути або порожнім, або бути вузлом, що має рядок і два спадкоємці.

Перевагою розмічених об'єднань є те, що вони збирають всі можливі варіанти в одному місці. При додаванні нових варіантів компілятор вказує міста, в яких слід переглянути реалізацію з урахуванням нових варіантів. Це запобігає несподівані помилки під час запуску програми.

Для компілятора розмічені об'єднання мають істотне значення, бо вони спрощують роботу з основоположною структурою даних в області компіляторів – абстрактними деревами синтаксису (abstract syntax tree, AST).

3.4 Зіставлення з шаблонами

Зіставлення з шаблонами (pattern matching) – це метод аналізу і обробки структур даних в мовах програмування, заснований на виконанні певних дій в залежності від збігу досліджуваного значення з тим чи іншим зразком. Зразком може бути константа, предикат або структура даних з довільною вкладеністю.

Покажемо використання зіставлення з шаблонами на прикладі. Припустимо, визначено бінарне дерево пошуку, як показано вище. Нам потрібно знайти дерево, яке має непорожнє ліве піддерево, і повернути значення цього піддерева.

Приклад такого дерева зображений на рис. 1.1.

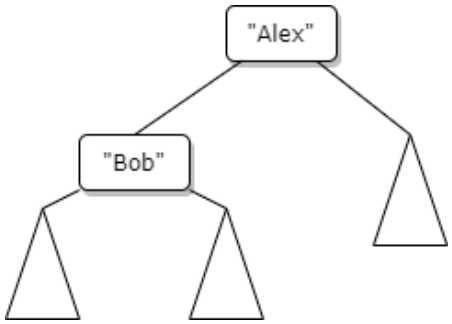

Рисунок 1.1 – Приклад бінарного дерева

Для виконання такої дії використовується такий код:

```
match tree with
| Node(_, Node(value, _, _), _) -> value
| _ -> "nothing found"
```

Тут використовується зіставлення з шаблоном структури даних з подвійною вкладеністю. Використання підкреслення (_) означає, що на його місці може бути все, що завгодно. Для реалізації такої дії на Java знадобилося б кілька разів використовувати оператор `instanceof` у зв'язці з приведенням типів.

Зіставлення з шаблонами разом з розміченими об'єднаннями є просто незамінними при написанні компіляторів, бо потрібно порівнювати структуру вихідного коду з заданими шаблонами.

3.5 Особливості системи типів F#

Ідея виведення типу полягає в тому, що вказівка типів для змінних і функцій не є обов'язковим, за винятком деяких випадків. Відсутність явної інформації про тип не означає, що F# є мовою з динамічною типізацією. F# – це ста-

тично типізована мова, оскільки компілятор визначає точний тип для кожної конструкції під час компіляції.

Автоматичне виведення типів дозволяє писати більш компактний код.

У списку параметрів вам не потрібно вказати тип кожного параметра. F# є статично типізованою мовою, і тому кожне значення та вираження має певний тип під час компіляції. Для тих типів, які ви не вказуєте явно, компілятор виводить тип, заснований на контексті. Якщо код використовує значення непослідовно, таким чином, що немає єдиного висновку, який задовольняє всім використанням значення, компілятор повідомляє про помилку.

Тип значення функції визначається на основі останнього виразу функції.

Наприклад, у коді нижче типи параметрів і тип значення функції виводиться як int, тому що константа 100 має тип int.

```
let f a b = a + b + 100
```

Є можливість явно зазначати тип функцій або змінних завдяки використанню анотацій типів.

Нерідко корисно додавати анотацію типів до параметра, коли потрібно використовувати поля або методи параметра:

```
let replace(str: string) =
    str.Replace("A", "a")
```

4 ЕТАПИ РОБОТИ КОМПІЛЯТОРУ

Задача компілятору полягає в створенні виконуваної програми із набору вихідних файлів. Для виконання цієї задачі компілятор проходить декілька фаз.

4.1 Обробка та компонування файлів

Розглянемо підхід, який використовується для обробки вхідних файлів і файлів бібліотеки.

На вході компілятор отримує набір вихідних файлів користувача. Також існує можливість підключення/відключення стандартної бібліотеки. Ця бібліотека містить часто використовувані функції, операції над списками, математичні операції, функції для роботи з файлами та інше. За замовчуванням, бібліотека включена в поточну програму.

Вихідні файли Scheme, які були надані користувачем та файли бібліотеки, перетворюються в файли асемблеру x86-64. У якості формату асемблеру використовується синтаксис AT&T (альтернативою до нього є синтаксис Intel).

До цих файлів асемблеру додається бібліотека часу виконання. Вона написана на мові C. Коротко розглянемо призначення файлів бібліотеки.

runtime.c – містить в собі функцію `main()`, з якої починається запуск програми. Спочатку виконується виділення пам’яті для стеку (stack) і кучі (heap), виконується підготовка збирачу сміття (garbage collector). Потім керування передається до згенерованого коду.

objects.c – містить помічні функції для роботи з об’єктами Scheme.

memory.c – містить функції виділення пам’яті і реалізацію збирача сміття. Функції виділення пам’яті викликаються зі сторони згенерованого коду кожен

раз при створенні нового об’єкту. Цей самий файл містить деякі глобальні дані, наприклад, вказівник на межі пам’яті.

Після цього всі файли асемблеру і код на мові С передається в компілятор GCC, який виконує остаточне компонування і виробляє виконуваний файл.

Зазначена послідовність дій зображена на рис. 4.1.

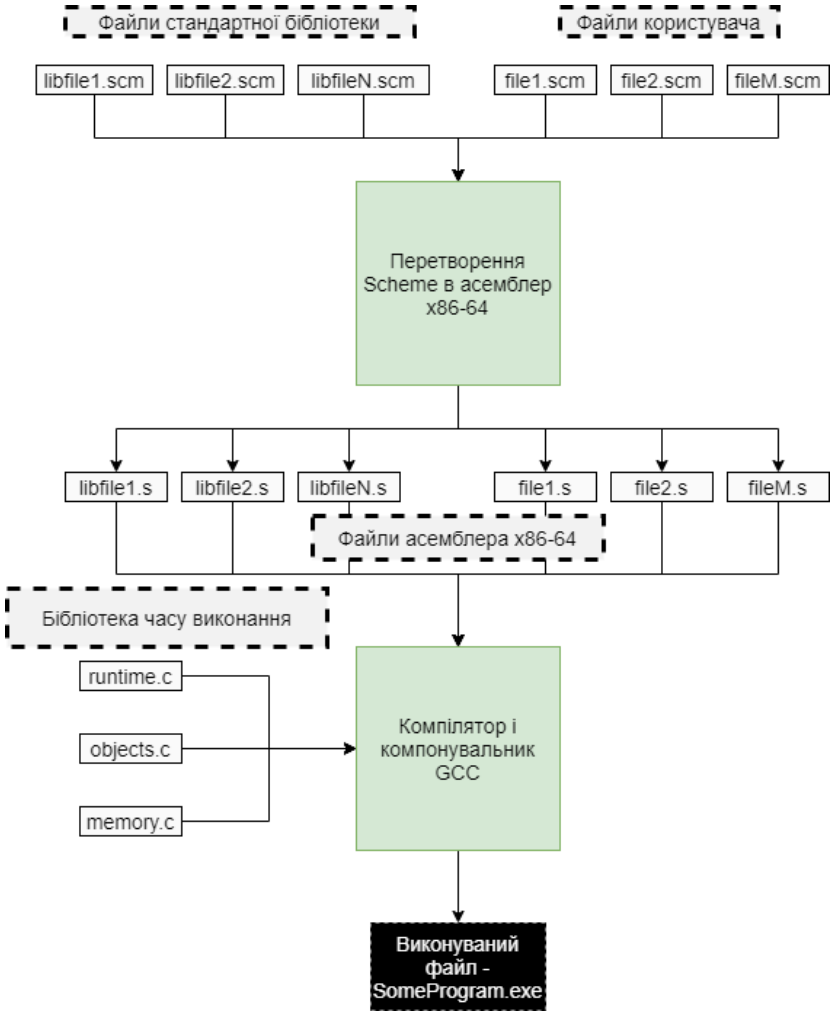

Рисунок 4.1 – Схема обробки файлів

Тепер розглянемо найбільш важливу та об’ємну частину компілятору – перетворення Scheme в асемблер.

4.2 Перетворення Scheme в асемблер

Перетворення коду Scheme в асемблер складається з послідовності фаз. Результат кожної фази є вхідними даними для наступної фази. Початковими вхідними даними є код Scheme. Кінцевим результатом є файл написаний на мові асемблеру згідно до рис. 4.2.

Рисунок 4.2 – Основні фази компілятору

Вихідний код представлений масивом символів. Для прикладу візьмемо послідовність символів, що показана на рис. 4.3.

(	+		a		7	3	)

Рисунок 4.3 – Послідовність символів

Першою фазою є лексичний аналіз. Лексичний аналіз – це перетворення послідовності символів в послідовність токенів (лексем) [2]. Аналогами токену в природній мові є знаки пунктуації та слова. Кожний токен являє собою елемент мови, наприклад ідентифікатор (`a`) або константу (`73`). Токен містить свій тип, а також може містити додаткову інформацію, таку як значення константи або ім’я ідентифікатора.

Послідовність токенів, що отримується із показаної на Рис. 3 послідовності символів, представлена на рис. 4.4.

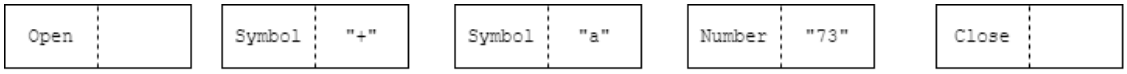

Рисунок 4.4 – Послідовність токенів

Як ми бачимо, деякі токени несуть додаткову інформацію, що була взята з вихідного коду. Для інших токенів така інформація є зайвою, тому що одного їх типу достатньо для визначення їх призначення.

Наступною фазою є синтаксичний аналіз, який аналізує послідовність токенів, групує їх згідно з правилами граматики і створює так звані s-вирази.

S-вираз – це рекурсивна структура даних, значення якої може бути або складеними (так звані комірки `Cons`), або простими, такими як символи і константи. Прості елементи також називають атомами. Комірки, на відміну від простих елементів, можуть містити в собі інші комірки і атоми.

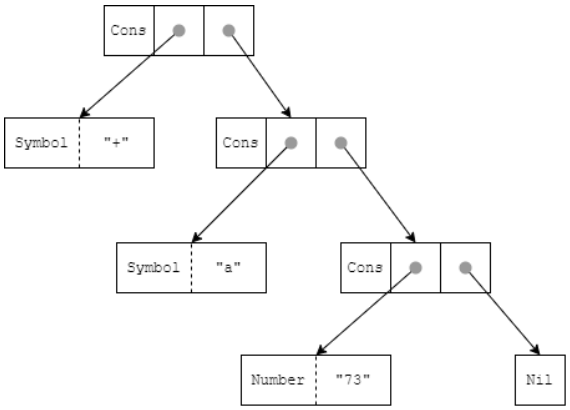

Рисунок 4.5 – Приклад s-виразу

На рис. 4.5 зображений результат лексичного аналізу послідовності токенів, що показана на рис. 4.4. Тут ми бачимо три комірки, кожна з яких містить символ або число. Дві перші посилаються на наступні комірки, а остання посилається на пустий список `Nil`. За значенням `Nil` визначається, що комірка є останньою в списку.

Після синтаксичного аналізу наступною фазою є так звана «децукоризація» (desugaring). Для розгляду цієї фази нам знадобиться наступне поняття.

Синтаксичний цукор (syntactic sugar) – це конструкції мови програмування, які спрощують для користувача роботу з мовою і які можуть бути реалізовані за допомогою вже наявних конструкцій.

Прикладом синтаксичного цукру в мові Java є for-each цикл, який дозволяє почергово виконувати дії над елементами послідовності. Цей цикл можна реалізувати за допомогою вже наявного в мові звичайного for циклу. Наприклад,

```
for (String s : words) {
     out.println(s);
}
```

перетворюється в:

```
for (Iterator<String> it = words.iterator(); it.hasNext(); ) {
     out.println(it.next());
}
```

Цей процес перетворення синтаксичного цукру в основні конструкції має назву децукоризації. Вона дозволяє зводити похідні конструкції до базових, що робить внутрішнє представлення мови більш компактним і зменшує кількість особливих випадків при реалізації компілятора.

Результатом роботи децукоризації є Core-форма. В цій формі складовим елементам мови надається семантичне значення. Наприклад, із s-виразів виділяються умовні вирази, застосування примітивних операцій, виклики функцій.

Над Core-формою виконуються додаткові перетворення. Одним з таких перетворювань є alpha-перейменування, які змінюють імена змінних таким чином, щоб кожна змінна мала унікальне ім’я. Ця властивість унікальності імен є дуже важливою у наступних фазах компіляції.

Отже, крім основних перетворювань між різними внутрішніми представленнями мови, існують ще і додаткові перетворювання форми в себе.

Потім Core-форма перетворюється в Intermediate-форму, тобто в проміжну форму. В Intermediate-формі кожна функція розбивається на блоки. Кожен блок складається з послідовності визначень змінних та завершується інструкцією переходу. Інструкція переходу може бути:

- повертання з функції;
- умовний перехід (if-then-else).

Розбиття на блоки спрощує задачу наступної оптимізації і підготовляє представлення коду до перетворення в асемблер.

Істотною відмінністю Intermediate-форми від Core-форми є те, що кожний складний вираз, який містить в собі вкладені підвирази, розбивається на прості вирази. Для цього вводяться додаткові локальні змінні. Цим змінним присвоюються значення простих виразів. Завдяки цьому підходу стає явною послідовність, за якою повинний обчислюватися кожний вираз.

Наприклад, вираз

```
(f (g x) (h y))
```

перетворюється у вираз

```
(let ((t1 (g x))
      (t2 (h y)))
  (f t1 t2))
```

В прикладі виклики функцій `g` і `h` зберігаються у тимчасові змінні `t1` і `t2`, які потім передаються до функції `f`.

Після цього над Intermediate-формою виконується перетворення, яке видаляє вкладені функції і робить їх глобальними. Мова Scheme дозволяє визначати функції всередині інших функцій. Внутрішні функції можуть захоплювати змінні із зовнішнього оточення. Такі змінні називаються вільними (free variables). При зазначеному перетворенні відбувається пошук вільних змінних, які потім додаються до функції як додаткові параметри. Після цього перетворення всі функції переміщуються до глобальної області видимості.

Наступною фазою є вибір інструкцій, також відома як генерація машинного коду. Результатом цієї фази є Machine-форма. Це внутрішнє представлення є найближчим до мови асемблеру. Ця форма являє собою послідовність інструкцій і позначок (labels). Кожна інструкція містить ім’я операції, і набір операндів.

Операнди можуть бути такими:

– тимчасові змінні,
– регістри процесора,
– адреса пам’яті.

На цій фазі кожна примітивна операція з Intermediate-форми розгортається у послідовність із декількох інструкцій.

Після вибору інструкцій відбувається перетворення внутрішнього представлення в код асемблера. Водночас відбувається перетворення констант у послідовність визначень констант асемблеру. Створюються таблиці, значення яких будуть відповідати значенням констант із коду Scheme. Також тут записуються імена глобальних змінних у вигляді рядку. Цей код записуються в файл асемблеру з розширенням `.s`, який потім буде переданий компілятору GCC.

Як ми бачимо, особливістю компілятора є те, що він проходить декілька фаз. Кожна фаза виробляє із вхідної форми наступну форму, що все ближче і ближче відповідає машинній мові.

З представлених вище фаз ми будемо детально розглядати фазу лексичного розбору, що створює послідовність токенів і фазу синтаксичного розбору, що перебудовує потік токенів в деревовидні s-вирази.

4.3 Обробка імен в компіляторі

Розглянемо відмінності традиційного підходу до обробки імен від підходу, що використовувався у нашому компіляторі.

Традиційно використовується структура даних, яка має назву таблиця імен [5]. Вона містить інформацію про змінні, функції, типи даних, області видимості. Приклад такої таблиці імен представлений на табл. 4.1.

Таблиця 4.1 – Приклад таблиці імен

Ім’я	Вид	Тип	Область видимості
`X`	Змінна	`Int`	Глобальна
`sqrt`	Функція	`float -> float`	Глобальна
`Y`	Змінна	`Double`	Локальна

Така таблиця є глобальною, тобто доступною для будь-якого компоненту компілятору. Вона заповнюється під час лексичного і синтаксичного аналізу і використовується на протязі наступних фаз компіляції.

У нашому компіляторі, зберігається лише інформація про глобальні змінні. Інформація про локальні змінні міститься в структурі даних, яка описує конкретну функцію. Таблиця глобальних змінних не є глобальною, а передається в структурі даних, яка відповідає кожній із фаз. Таким чином, кожна фаза отримує стільки інформації, скільки їх потрібно.

Нагадаємо, що Scheme дозволяє використовувати ідентифікатори зі спеціальними символами. Такі ідентифікатори змінюються таким чином, щоб вони могли бути представлені в мові асемблера. Наприклад `pair?` перетворюється в `pairQmark`, `read-char` в `readMinuschar`. Таблиця змінних зберігає обидва імені початкове і змінене.

Крім цього, зберігається інформація про константи. В начальних фазах компіляції збирається інформація про числові і рядкові константи. Кожна константа отримує унікальне ім’я, за яким зберігається саме значення константи. Ці дані передаються між фазами, подібно до того, як передається таблиця глобальних змінних. У кінцевій фазі інформація про константи використовується в процесі запису Machine-форми у файл асемблеру.

5 ДОКЛАДНИЙ ОПИС ЛЕКСЕМ

Наведемо граматику для опису лексем.

```
<lexem> ::=
    <openBracket> |
    <closeBracket> |
    <openParen> |
    <closeParen> |
    <dot> |
    <quote> |
    <unquote> |
    <boolean> |
    <character> |
    <string> |
    <number> |
    <floatNumber> |
    <identifier>
```

Кожна лексема може бути спеціальним символом, таким як <openParen>, або константою <floatNumber>, або ідентифікатором <identifier>.

Найпростішими є правила для спеціальних символів.

```
<openBracket> ::= "["
<closeBracket> ::= "]"
<openParen> ::= "("
<closeParen> ::= ")"
<dot> ::= "."
<quote> ::= "'"
<unquote> ::= ","
```

Ці правила просто шукають відповідний символ у вхідній послідовності.

Для булевих констант використовується наступне правило:

```
<boolean> ::= "#t" | "#T" | "#f" | "#F"
```

Константи `#t` і `#T` відповідають значенню «true», `#f` і `#F` – «false».

Правило для символьних констант (character):

```
<character> ::=
    "\#" <будь-який символ> |
    "\#newline" |
    "\#space"
```

Перший варіант означає, що після послідовності символів # і \ обирається будь-який наступний символ. `#\newline` відповідає символу нового рядку, а `#\space` символу пробілу. Приклади символьних констант:

```
#\a #\z #\! #\8
```

Правило для рядків:

```
<string> ::= "\"" <будь-який символ крім "\"">* "\""
```

Це правило означає, що рядок складається з пари подвійних лапок, між якими може бути від нуля і більше будь-яких символів, крім символу подвійних лапок. Символ зірки (*) в правилі вказує на множинність від нуля і більше. Для використання в рядку символу подвійних лапок (") необхідно перед ним додати символ зворотного слешу (\). Приклади рядків:

```
"string 1"
"one \"two\" three"
```

Цілочисельний константи описуються таким правилом:

```
<number> ::= <sign> <digits>
```

Тут використовуються такі допоміжні правила:

```
<sign> ::= "+" | "-" | <empty>
<digits> ::= <digit>+
<digit> ::= "0" ... "9"
<empty> ::= ""
```

Правило <empty> відповідає вхідній послідовності нульової довжини.

<digit> відповідає одній з десяти цифр, а <digits> складається з послідовності з однієї і більше цифр.

<sign> відповідає символу плюс (+), або мінус (-), або жодному з них.

Таким чином, константа цілого числа складається з послідовності одного і більше цифр, перед якими може стояти плюс (+) або мінус (-). Приклади цілих чисел:

```
3 +881 -2098
```

Правило для чисел з рухомою точкою:

```
<floatNumber> ::=
    <sign> <digits> <dot> <exponentOpt> |
    <sign> <digits> <dot> <digits> <exponentOpt> |
    <sign> <dot> <digits> <exponentOpt> |
    <sign> <digits> <exponentOpt>
```

Допоміжні правила:

```
<exponentOpt> ::= <exponent> | <empty>
<exponent> ::= ("e" | "E") <sign> <digits>
```

<exponentOpt> говорить про можливу частини експоненті в числі. Експоненціальна частина складається з символу "e" або "E", за яким може слі-

дувати знак плюс (+) або мінус (-), потім послідовність цифр. Приклади чисел з рухомою точкою:

```
+321.e34 -111.111 -.123e-20 9393e3
```

Останнім видом лексем є ідентифікатор:

```
<identifier> ::=
    <initial> <subsequent>* |
    "+" | "-" | "..."

<initial> ::=
    <letter> | "!" | "$" | "%" | "&" |
    "*" | "/" | ":" | "<" | "=" | ">" | "?" | "~" |
    "_" | "^" | "@"

<subsequent> ::=
    <initial> | <digit> | <dot> | "+" | "-"
```

Ідентифікатор не може починатися з символу, з якого може починатися число, тобто з цифри, знаку плюс (+), мінус (-) або точки (.). Винятком з цього правила є три ідентифікатори: +, -, ...

6 ОПИС ТОКЕНІВ

Перш ніж перейти до опису реалізації лексем – токенів, розглянемо поняття розмічених об'єднань.

Розмічене об'єднання (discriminated union) - це тип, який визначається користувачем. Значення цього типу можуть бути одним з іменованих варіантів, при цьому можливо, що ці варіанти мають різну кількість і типи полів. Розмічені об'єднання також підтримують рекурсивні визначення, дозволяючи з легкістю визначати такі структури як дерева та інші рекурсивні дані. Для реалізації токенів ми використовували розмічене об’єднання.

Оголошення типу даних Token на мові F# виглядає наступним чином:

```
type Token =
    | Open
    | Close
    | OpenBr
    | CloseBr
    | Dot
    | Quote
    | Unquote
    | Number of string
    | FloatNum of string
    | String of string
    | Symbol of string
    | Bool of bool
    | Char of char
```

В табл. 6.1 показана відповідність лексем і токенів.

Таблиця 6.1 – Лексеми та відповідні їм токени

Лексема	Токен	Опис	Приклад
1	2	3	4
<openBracket>	OpenBr	Відкриваюча квадратна дужка	[
<closeBracket>	CloseBr	Закриваюча квадратна дужка	]
<openParen>	Open	Відкриваюча кругла дужка	(

Продовження таблиці 6.1

1	2	3	4
<closeParen>	Close	Закриваюча кругла дужка	)
<dot>	Dot	Точка	.
<quote>	Quote	Лапка	'
<unquote>	Unquote	Відмінена лапка	,
<boolean>	Bool	Булева константа	#t
<character>	Char	Символьна константа	#\?
<string>	String	Строкова константа	"Great news"
<number>	Number	Константа цілого числа	81883
<floatNumber>	FloatNum	Константа числа з рухомою точкою	3.141516
<identifier>	Symbol	Ідентифікатор, використовується для імен, ключових слів і символів (symbols)	send-message

Відкриваючим і закріваючим круглим дужкам відповідають токени `Open`, `Close`, `OpenBr`, `CloseBr`. Інші спеціальні токени: `Dot`, `Quote`, `Unquote`.

Ці токени не мають полів, бо не потребують додаткових даних. Для їх використання досить визначити варіант токена за ім'ям. Наприклад, на рис. 6.1 показаний токен `Open`.

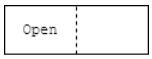

Рисунок 1.1 – Токен Open

`Number` і `FloatNum` – числові токени. Обидва варіанти містять рядок в як єдине поле. У цьому рядку зберігається текстовий вид числа, який був зчитаний з вихідного коду Scheme. Обробка рядку і конвертація рядку в числові типи

`int` і `double` буде проходити пізніше, на етапі синтаксичного розбору. На рис. 6.2 зображений приклад токену `Number`.

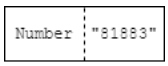

Рисунок 6.2 – Токен Number

Токени `String` и `Symbol` також містять рядок як поле. На рис. 6.3 представлений токен `Symbol`.

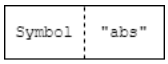

Рисунок 6.3 – Токен Symbol

Токен `Bool` має логічне поле, що містить значення булевої константи (рис. 6.4).

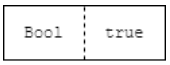

Рисунок 6.4 – Токен Bool

І, нарешті, токен `Char` містить значення символьної константи (рис. 6.5).

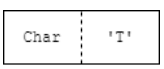

Рисунок 6.5 – Токен Char

7 ОПИС АЛГОРИТМУ ЛЕКСИЧНОГО РОЗБОРУ

Завдання алгоритму лексичного розбору полягає в перетворенні послідовності символів в послідовність токенів.

На кожній ітерації циклу розбору містяться такі дані: вхідна послідовність символів і список вже розібраних токенів.

Кроки алгоритму:

1. Якщо поточний символ – символ пробілу, нового рядка або табуляції, то він пропускається і виконується перехід до наступного символу.
2. Якщо поточний символ – крапка з комою (`;`), то пропускаються всі символи від крапки з комою до символу нового рядка включно. Таким чином пропускаються коментарі.
3. Якщо поточний символ – один зі спеціальних символів (`() [],`), то додаємо в список відповідний токен і переходимо до наступного символу.
4. Якщо поточний символ – подвійні лапки (`"`), то починаємо розбір рядкової константи. Запам'ятовуємо всі символи до закриваючих подвійних лапок, додаємо токен `String` в список, переходимо до наступного символу після подвійних лапок.
5. Якщо у вхідній послідовності наступні два символи рівні Булевим константам (`#t #T #f #T`), то додаємо в список токен `Bool` з відповідним значенням і переходимо до наступного символу після константи.
6. Якщо наступні символи – `#\space` або `#\newline`, то додаємо токен `Char` з відповідним значенням константи і переходимо до наступного символу.
7. Якщо перші два символи – знак гратки (`#`) і зворотний слеш (`\`), то зчитується третій символ та додається токен `Char`, який містить третій символ. Потім переходимо до наступного символу.

8. Якщо перший символ – цифра, зчитуємо ціле число або число з рухомою точкою і додаємо `Number` або `FloatNum`. Потім переходимо до наступного символу.
9. Якщо перший символ – плюс (+) або мінус (-), а другий – цифра, то зчитуємо число як у попередньому пункті.
10. Якщо перший символ – точка (.), а другий – цифра, то зчитуємо число с рухомою точкою и додаємо токен `FloatNum`.
11. Якщо вхідна послідовність ще не є пустою – зчитуємо символ (symbol), доки не зустрінеться роздільник. Роздільником символів є пробіл, табуляція, новий рядок, дужки (( ) [ ]), подвійні лапки("). Додаємо токен `Symbol`.
12. Якщо вхідна послідовність є пустою – повертаємо список отриманих токенів.

8 ОПИС СТРУКТУРИ S-ВИРАЗІВ

S-вираз – це рекурсивна структура даних. Кожний s-вираз може бути:

– атомом, до яких відносяться константи та символи;

– формою, що являють собою списки, які містять інші атоми або форми.

У мові Lisp і його діалектах s-вирази використовуються для подання коду і для представлення даних.

S-вирази часто порівнюють з форматом XML. Головна відмінність полягає в тому, що s-вирази мають тільки одну форму включення, впорядковану пару, а теги XML можуть містити атрибути та інші теги. Тому розбір s-виразів є простішим завданням, ніж розбір XML.

S-вирази створюються під час синтаксичного аналізу з послідовності токенів та мають наступне подання:

```
type SExpr =
    | Cons of SExpr * SExpr
    | Nil
    | Number of int
    | FloatNumber of float
    | String of string
    | Char of char
    | Symbol of string
    | Bool of bool
    | VoidValue
```

Відзначимо, що в s-виразах ще не виявлені такі мовні конструкції як умовні вирази, привласнення тощо. Тут всі форми присутні поки в «сирому» вигляді. Основне призначення s-виразів в тому, що вони групують і впорядковують простіші елементи.

Розглянемо, які є види s-виразів.

Таблиця 8.1 – S-вирази та відповідні їм токени

S-вираз	Короткий опис	Аналогічний токен
`Cons`	Пара s-виразів	`---`
`Nil`	Пустий список	`Nil`
`Number`	Константа цілого числа	`Number`
`FloatNumber`	Константа числа с рухомою точкою	`FloatNum`
`String`	Рядкова константа	`String`
`Char`	Символьна константа	`Char`
`Symbol`	Символ	`Symbol`
`Bool`	Булева константа	`Bool`
`VoidValue`	Невизначене значення	`---`

З табл. 8.1 видно, що для більшості видів s-виразів є токени, яким вони відповідають. Основна відмінність s-виразів від послідовності токенів є те, що s-вирази є деревовидною структурою даних. Також, в s-виразах немає знаків пунктуації, таких як відкриваючи та закриваючи дужки, бо вони використовується для створення значень `Cons`.

`Cons` (від construct - конструювання) являє собою вираз, який містить два вкладених s-вирази. У Scheme є однойменна функція `cons`, яка конструює об'єкт, що містить два значення, в пам'яті.

`Nil` є порожній список. У вихідному коді Scheme він представлений послідовністю символів () або []. В процесі обробки списків по цьому елементу визначається, що досягнутий кінець списку.

`VoidValue` не має відповідного представлення в вихідному коді. Це проміжне значення, яке використовується на фазі децукоризації. Річ в тому, що кожен вираз має мати значення. Якщо останнім виразом у функції є присвоювання, наприклад

```
(set! a 3)
```

то результатом роботи функції буде якраз `VoidValue`.

Інші елементи s-виразів мають аналогічні значення в множині токенів. Відмінністю є те, що токени `Number` і `FloatNum` містять значення типу `string`, які були отримані з вихідного коду, а s-вирази `Number` і `FloatNumber` вже містять числові значення `int` і `float`.

Розглянемо приклади s-виразів.

Нижче показаний список складається з трьох символів `a`, `b` і `c`:

```
(a b c)
```

На рис. 8.1 зображений цей список.

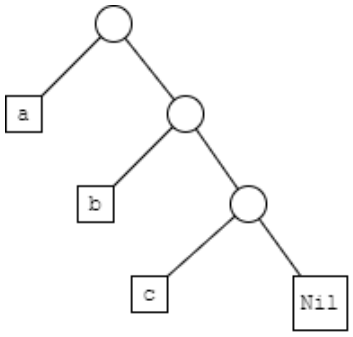

Рисунок 8.1 – S-вираз для простого списку

Тут ми бачимо три елементи `Cons`, які зображені колом. Кожен з них містить по два значення. `Nil` вказує, що `Cons`, в якому він міститься, є останнім.

Складніший приклад, який містить вкладені списки:

```
(* 2 (+ 3 4))
```

На Рис. 12 показаний s-вираз, який відповідає цьому прикладу.

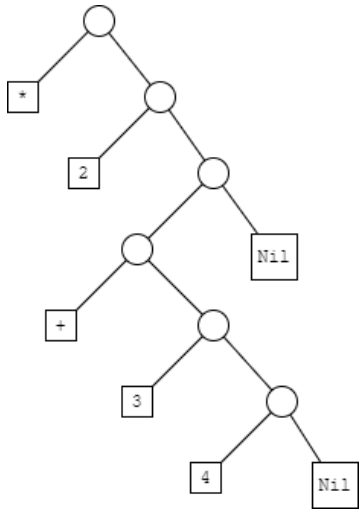

Рисунок 8.2 – Вкладені списки

З рис. 8.2 видно, що `Cons` елементи можуть містити інші `Cons` елементи. Таким чином, виходить вкладеність s-виразів. У цьому прикладі присутні 6 елементів `Cons`. Наявність двох елементів `Nil` вказує на те, що тут використовується два списки, один з цих списків вкладений в інший.

9 ОПИС АЛГОРИТМУ СИНТАКСИЧНОГО РОЗБОРУ

Алгоритм синтаксичного розбору приймає на вхід послідовність токенів і створює s-вираз, який відповідає цій послідовності.

В алгоритмі використовується три процедури:

1. `parseExpr` –процедура, що керує.
2. `parseNonlist` – розбирає атомарні елементи.
3. `parseList` – розбирає список елементів.

Кожна з цих процедур приймає список токенів. Ці процедури повертають пару значень: створений s-вираз та список токенів, що залишились. Цей список повертається для того, щоб процедура, яка викликає, могла знайти місце, з якого можна продовжувати розбір.

Кроки роботи процедури `parseExpr`:

1. Якщо перший токен – `Open` або `OpenBr`, то викликаємо процедуру `parseList` і передаємо відповідну відкриваючу дужку.
2. Інакше викликаємо процедуру `parseNonlist`.

Кроки роботи процедури `parseNonlist`:

1. Якщо перший токен – `Quote`, то розбираємо наступний вираз і обертаємо його в форму `(quote …)`.
2. Якщо перший токен – `Unquote`, то розбираємо наступний вираз і обертаємо його в форму `(unquote …)`.
3. Якщо перший токен – `Number` або `FloatNum`, то розбираємо рядок, із рядка створюємо числове значення і повертаємо відповідний s-вираз: `Number` або `FloatNumber`.
4. Якщо перший токен – `Symbol`, `String`, `Bool` або `Char`, то повертаємо аналогічний s-вираз.

Процедура `parseList` відрізняється від інших тим, що додатково приймає тип відкриваючою дужки (квадратна `[` або кругла `(`). Ця процедура містить список s-виразів, які вже були створені на поточному рівні. Ми будемо називати цей список акумулятором.

Кроки роботи процедури `parseList`:

1. Якщо поточний токен – `Dot`, то викликаємо процедуру `parseExpr`. В токенах, що залишились, першим повинен бути токен закриваючої дужки. Формуємо і повертаємо крапковий список. Це особливий вид списку.
2. Якщо поточний токен – `Close`, то перевіряємо, що відкриваюча дужка на даному рівні дорівнює `Open`. Якщо це так, то додаємо `Nil` в якості останнього елементу списку і повертаємо список.
3. Якщо поточний токен – `Open`, то рекурсивно викликаємо `parseList`, передаємо `Open` як відкриваючу дужку. Додаємо список в акумулятор і продовжуємо розбирати токени, які залишились.
4. Якщо поточний токен – `CloseBr`, то перевіряємо, що відкриваюча дужка на даному рівні дорівнює `OpenBr`. Якщо це так, то додаємо `Nil` в якості останнього елементу списку і повертаємо список.
5. Якщо поточний токен – `OpenBr`, то рекурсивно викликаємо `parseList`, передаємо `OpenBr` як відкриваючу дужку. Додаємо список в акумулятор і продовжуємо розбирати токени, які залишились.
6. Інакше викликаємо процедуру `parseNonlist`, додаємо результат в список, і продовжуємо розбирати токени, які залишились.

На рис. 9.1 показаний алгоритм роботи процедури `parseList`.

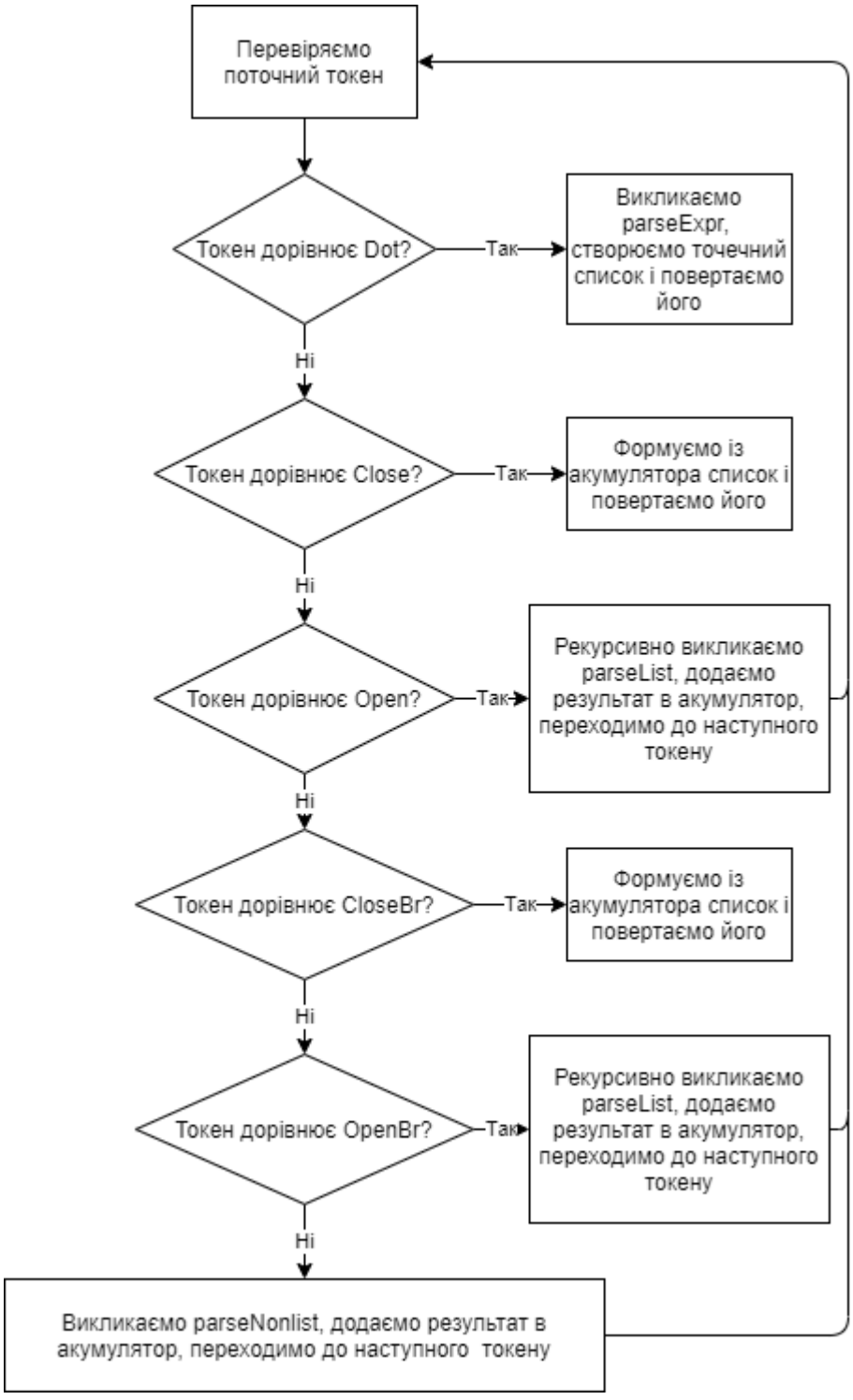

Рисунок 9.1 – Алгоритм роботи процедури parseList

10 ПРИКЛАДИ

10.1 Приклад 1. Обчислення абсолютного значення

10.1.1 Вихідний код

```
(define (abs x)
  (if (< x 0)
    (- 0 x)
    x))

(abs -331)
```

У цьому невеликому прикладі представлена функція abs, яка повертає абсолютне значення аргументу. Тіло функції складається з умовного виразу. Якщо аргумент має від'ємне значення, то повертається число з протилежним знаком. Інакше повертається аргумент без змін.

10.1.2 Результат лексичного аналізу

Результатом синтаксичного аналізу буде послідовність токенів:

```
Open
Symbol "define"
Open
Symbol "abs"
Symbol "x"
Close
Open
Symbol "if"
Open
Symbol "<"
Symbol "x"
Number "0"
Close
Open
Symbol "-"
Number "0"
Symbol "x"
Close
Symbol "x"
```

```
Close
Close
Open
Symbol "abs"
Number "-331"
Close
```

10.1.3 Результат синтаксичного аналізу

```
Cons
- Symbol "define"
- Cons
 - Cons
  - Symbol "abs"
  - Cons
   - Symbol "x"
   - Nil
 - Cons
  - Cons
   - Symbol "if"
   - Cons
    - Cons
     - Symbol "<"
     - Cons
      - Symbol "x"
      - Cons
       - Number 0
       - Nil
    - Cons
     - Cons
      - Symbol "-"
      - Cons
       - Number 0
       - Cons
        - Symbol "x"
        - Nil
     - Cons
      - Symbol "x"
      - Nil
  - Nil
```

10.1.4 Результат запуску програми

Ми запускаємо компілятор за допомогою інструменту dotnet run. Аргумент `-p` задає шлях до проекту. Потім знаходяться аргументи до самого компі-

лятору. Аргумент `.\examples\abs.scm` вказує шлях до вихідного файлу Scheme, який буде компілюватися. Аргумент `-o` задає шлях до файлу програми, яка буде створена в результаті.

Після виконання команди створюється файл `abs.exe` в папці `misc`.

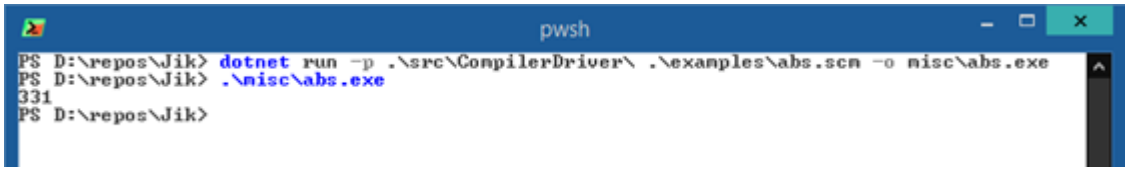

Рисунок 10.1 – Компіляція і запуск програми abs.exe

На рис. 10.1 показаний результат – абсолютне значення числа `-331`, тобто `331`.

10.2 Приклад 2. Сортування злиттям

10.2.1 Вихідний код

```
 (define (split ls)
  (letrec ([split-h (lambda (ls ls1 ls2)
                      (cond
                        [(or (null? ls) (null? (cdr ls)))
                         (cons (reverse ls2) ls1)]
                        [else (split-h (cddr ls)
                                       (cdr ls1) (cons (car ls1) ls2))]))
])
    (split-h ls ls '())))

(define (merge pred ls1 ls2)
  (cond
    [(null? ls1) ls2]
    [(null? ls2) ls1]
    [(pred (car ls1) (car ls2))
     (cons (car ls1) (merge pred (cdr ls1) ls2))]
    [else (cons (car ls2) (merge pred ls1 (cdr ls2)))]))

(define (merge-sort pred ls)
  (cond
```

```
    [(null? ls) ls]
    [(null? (cdr ls)) ls]
    [else (let ([splits (split ls)])
            (merge pred
              (merge-sort pred (car splits))
              (merge-sort pred (cdr splits))))]))

(merge-sort (lambda (x y) (< x y)) '(9 6 7 3))
```

У цій програмі показується реалізація сортування зв'язного списку злиттям.

Ідея алгоритму полягає в тому, що вхідний список елементів поділяється на два, потім рекурсивно сортується ці два списки, потім результат об'єднується в один сортований список.

Функція `split` розділяє список на половину, і повертає два списки. Функція `merge` з'єднує два відсортовані списки в один відповідно до переданого предикату `pred`. Функція `merge-sort` приймає предикат `pred` і список `ls` і повертає відсортований список. Після визначення функцій виконується сортування списку `(9 6 7 3)`. В якості предиката передається лямбда-функція, яка порівнює два числа.

На цьому прикладі видно як відбувається в Scheme робота з основним типом даними - зв'язковими списками. Функція `null?` перевіряє чи є список порожнім. Функція `cons` формує новий список, додаючи до колишнього списку один елемент. Функція `car` повертає голову списку (перший елемент), а функція `cdr` - хвіст списку (той самий список, тільки без першого елемента). Робота алгоритму зображена на рис. 10.2.

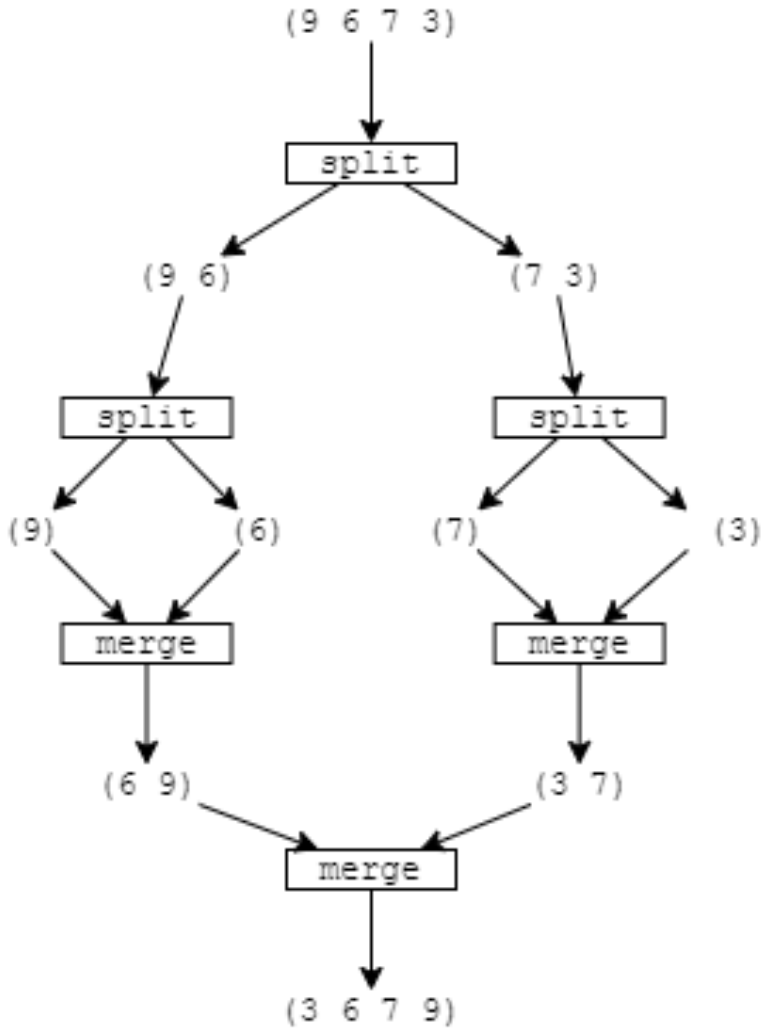

Рисунок 10.2 – Робота алгоритму сортування злиттям

10.2.2 Результат лексичного аналізу

Нижче представлений результат лексичного аналізу для визначення функції `split`.

Open Symbol "define" Open Symbol "split" Symbol "ls"	Symbol "null?" Open Symbol "cdr" Symbol "ls" Close	Open Symbol "cons" Open Symbol "car" Symbol "ls1"

```
Close
Open
Symbol "letrec"
Open
OpenBr
Symbol "split-h"
Open
Symbol "lambda"
Open
Symbol "ls"
Symbol "ls1"
Symbol "ls2"
Close
Open
Symbol "cond"
OpenBr
Open
Symbol "or"
Open
Symbol "null?"
Symbol "ls"
Close
Open
```

```
Close
Close
Open
Symbol "cons"
Open
Symbol "reverse"
Symbol "ls2"
Close
Symbol "ls1"
Close
CloseBr
OpenBr
Symbol "else"
Open
Symbol "split-h"
Open
Symbol "cddr"
Symbol "ls"
Close
Open
Symbol "cdr"
Symbol "ls1"
Close
```

```
Close
Symbol "ls2"
Close
Close
CloseBr
Close
Close
CloseBr
Close
Open
Symbol "split-h"
Symbol "ls"
Symbol "ls"
Quote
Open
Close
Close
Close
Close
```

10.2.3 Результат синтаксичного аналізу

Нижче представлений результат лексичного аналізу функції `split`.

```
Cons
- Symbol "define"
- Cons
 - Cons
  - Symbol "split"
  - Cons
   - Symbol "ls"
   - Nil
 - Cons
  - Cons
   - Symbol "letrec"
   - Cons
    - Cons
     - Cons
      - Symbol "split-h"
      - Cons
       - Cons
        - Symbol "lambda"
        - Cons
         - Cons
          - Symbol "ls"
          - Cons
```

```
          - Symbol "ls1"
          - Cons
           - Symbol "ls2"
           - Nil
        - Cons
         - Cons
          - Symbol "cond"
          - Cons
           - Cons
            - Cons
             - Symbol "or"
             - Cons
              - Cons
               - Symbol "null?"
               - Cons
                - Symbol "ls"
                - Nil
              - Cons
               - Cons
                - Symbol "null?"
                - Cons
                 - Cons
                  - Symbol "cdr"
                  - Cons
                   - Symbol "ls"
                   - Nil
                 - Nil
```

10.2.4 Результат запуску програми

Аналогічно попередньому прикладу, тут ми компілюємо вихідний файл `examples\msort.scm`, зберігаємо додаток в файлі `misc\msort.exe` і запускаємо цю програму. В результаті на рис. 10.3 ми бачимо відсортований список `(3 6 7 9)`.

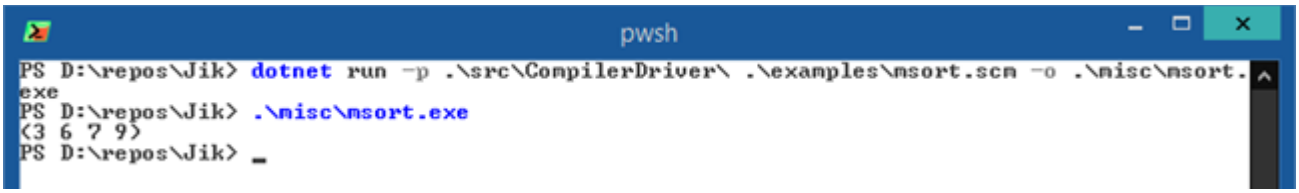

Рисунок 10.3 – Компіляція і запуск програми msort.exe

10.3 Приклад 3. Порівняння інваріантів мереж Петрі

Вихідний код програми доступний у додатках.

Завдання полягає в порівнянні набору інваріантів з заданим зразком. Кожен інваріант складається з множини імен. Кожне ім'я може являти собою перехід або фішку (позицію).

Інваріанти становлять собою потужний інструмент для дослідження властивостей мереж Петрі. За допомогою інваріантів можливо перевіряти мережеві протоколи [8].

Програма, що описуються у цьому прикладі, була використана для перевірки обчислювальних решіток [9].

10.3.1 Алгоритм порівняння

Розглянемо алгоритм порівняння інваріантів:

- прочитати текст кожного з файлів;
- розділити текст у рядки;
- відфільтрувати зайві рядки;
- розділити кожен рядок на імена і додати до списку;
- відсортувати імена;
- відсортувати лінії;
- порівняти довжину списків;
- порівняти списки за елементами.

Для сортування використовувався алгоритм сортування злиттям.

10.3.2 Формат запису інваріантів

Наведемо приклад запису інваріантів мереж Петрі:

```
{pb_0,-1^,0,0} {pb_0,1^,0,0} {pb_1,-1^,0,0} {pb_1,1^,0,0} {pb1^,0,0}
```

У цьому прикладі ми бачимо інваріант, що складається з п’яти фішок. Кожен файл, що подається на перевірку, має один і більше інваріантів схожого виду. Інваріанти можуть мати різну кількість елементів.

ВИСНОВКИ

В роботі була представлена реалізація компонентів компілятора для функціональної мови програмування Scheme. Були показані призначення і загальна структура компілятора. Компілятор складається з послідовності фаз, кожна з яких виконує певне завдання і перетворює внутрішнє представлення коду. Кінцевою метою компілятора є збірка нової програми.

Для докладного опису ми вибрали два компоненти компілятора: лексичний і синтаксичний аналізатори. Лексичний аналізатор займається перетворенням послідовності символів в послідовність токенів. Токени містять в собі інформацію про лексичні елементи мови, які називаються лексемами. Алгоритм лексичного аналізу побудований відповідно до правил граматики лексем.

Метою синтаксичного аналізатора є перетворення послідовності токенів в s-вирази. S-вирази – це деревоподібна структура даних, яка є допоміжною структурою для подальшої побудови абстрактного дерева синтаксису.

В кінці роботи ми показали приклади, на яких продемонстровано основні підходи побудови програм на мові Scheme. На цих же прикладах були показані результати роботи лексичного і синтаксичного аналізаторів.

Програми на мовах функціонального програмування використовують за останніх часів у якості атрибутів графічних елементів сітей Петрі, які застосовуються для моделювання телекомунікаційних систем та мереж, що підвищує практичну цінність результатів роботи. Розроблений компілятор планується до використання як компонент спеціалізованої системи моделювання телекомунікаційних мереж.

Результати роботи були представлені на 75-й науково-технічній конференції професорсько-викладацького складу, науковців, аспірантів та студентів ОНАЗ ім. О. С. Попова 11-15 грудня 2020 року [10].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Dybvig R.K. The Scheme Programming Language. Third Edition. : The MIT Press, 2003. 504 p.
2. Nystrom R. Crafting Interpreters. URL: https://craftinginterpreters.com/. (дата звернення 23.11.2020).
3. Syme D., Granicz A., Cisternino A. Expert F# 4.0. : Apress 2015. 582 p.
4. Большакова Е.И., Груздева Н.В. Основы программирования на языке Лисп: учебное пособие. – М.: Издательский отдел факультета ВМК МГУ имени М.В.Ломоносова; МАКС Пресс, 2010 – 112 с.
5. Хантер Р. Проектирование и конструирование компиляторов. М.: Финансы и статистика, 1984
6. McCarthy J. Recursive functions of symbolic expressions and their computation by machine. Communications of the ACM, 1960, Vol. 3, N 4, p. 184-195.
7. IEEE Computer Society. IEEE Standard for the Scheme Programming Language, May 1991. IEEE Std 1178-1990.
8. Zaitsev D.A. Clans of Petri Nets: Verification of protocols and performance evaluation of networks, LAP LAMBERT Academic Publishing, 2013, 292 p.
9. Zaitsev D.A., Shmeleva T.R., Guliak R.N., “Analyzing Multidimensional Communication Lattice with Combined Cut-through and Store-and-forward Switching Node,” Lect. Notes in Networks, Syst., Vol. 201, 2021, Raghvendra Kumar et al. (Eds): Next Generation of Internet of Things, 978-981-16-0665-6, 498624_1_En, (Chapter 58).
10. Гуляк Р.М. Разработка компилятора языка Scheme для встраивания в моделирующие системы // Матеріали 75-ї науково-технічної конференції професорсько-викладацького складу, науковців, аспірантів та студентів (м. Одеса, 11-15 грудня 2020 року). 2020 р. С. 88-90.

ДОДАТОК А ВИХІДНИЙ КОД

А.1 Вихідний код лексичного аналізатору

```
open System
open System.Text

type Token =
    | Open
    | Close
    | OpenBr
    | CloseBr
    | Dot
    | Quote
    | Unquote
    | Number of string
    | FloatNum of string
    | String of string
    | Symbol of string
    | Bool of bool
    | Char of char

let specialChars = [
    "tab", '\t'
    "newline", '\n'
    "return", '\n'
    "space", ' '
]

let charsToString rev cs =
    let cs : seq<char> = if rev then Seq.rev cs else cs
    cs |> Seq.toArray |> System.String.Concat

let rec readInt acc cs =
    match cs with
    | c :: cs' when Char.IsDigit(c) ->
        readInt (c :: acc) cs'
    | _ -> acc, cs

let readExponent acc cs =
    match cs with
    | 'e' :: cs | 'E' :: cs ->
        match cs with
        | c :: cs when c = '+' || c = '-' ->
            readInt (c :: 'e' :: acc) cs
        | _ ->
```

```
            readInt ('e' :: acc) cs
    | _ -> acc, cs

let readNumber sign cs =
    let acc = Option.toList sign
    match cs with
    | c :: _ when Char.IsDigit(c) ->
        let acc, cs = readInt acc cs
        match cs with
        | '.' :: cs ->
            let acc = '.' :: acc
            let acc, cs = readInt acc cs
            let acc, cs = readExponent acc cs
            FloatNum(charsToString true acc), cs
        | 'e' :: _ | 'E' :: _ ->
            let acc, cs = readExponent acc cs
            FloatNum(charsToString true acc), cs
        | _ ->
            Number(charsToString true acc), cs
    | '.' :: cs ->
        let acc = '.' :: acc
        let acc, cs = readInt acc cs
        let acc, cs = readExponent acc cs
        FloatNum(charsToString true acc), cs
    | _ -> failwith "Not a number"

let tokenize source =
    let isSymbolChar c =
        match c with
        | '(' | ')' | '[' | ']' -> false
        | c when Char.IsWhiteSpace(c) -> false
        | _ -> true

    let isEscapedChar c =
        match c with
        | '\\' | '"' -> true
        | _ -> false

    let rec eatString (acc:StringBuilder) = function
        | '\\' :: c :: cs when isEscapedChar c ->
            acc.Append(c) |> ignore
            eatString acc cs
        | '"' :: cs -> acc.ToString(), cs
        | c :: cs ->
            acc.Append(c) |> ignore
```

```
            eatString acc cs
        | [] -> failwith "EOF not espected"

    let character cs =
        let found =
            List.tryPick (fun (s, c) ->
                let l = Seq.length s
                if Seq.toList s = List.truncate l cs then
                    Some (c, (List.skip l cs))
                else None) specialChars
        match found, cs with
        | Some (c, cs'), _ -> Char c, cs'
        | _, (c::cs') -> Char c, cs'
        | _ -> failwith "tokenize: character: EOF not expected"

    let rec number (acc:string) = function
        | d :: cs when Char.IsDigit(d) ->
            number (acc + d.ToString()) cs
        | cs -> acc, cs

    let rec symbol (acc:string) = function
        | d :: cs when isSymbolChar d ->
            symbol (acc + d.ToString()) cs
        | cs ->
            acc, cs

    let rec skipUntilNewline = function
        | '\n' :: cs | '\r' :: '\n' :: cs | '\r' :: cs ->
            cs
        | _ :: cs -> skipUntilNewline cs
        | [] -> []

    let rec loop acc cs =
        match cs with
        | w :: cs when System.Char.IsWhiteSpace(w) -> loop acc cs
        | ';' :: cs -> loop acc (skipUntilNewline cs)
        | '(' :: cs -> loop (Open :: acc) cs
        | ')' :: cs -> loop (Close :: acc) cs
        | '[' :: cs -> loop (OpenBr :: acc) cs
        | ']' :: cs -> loop (CloseBr :: acc) cs
        | '"' :: cs ->
            let s, cs' = eatString (StringBuilder()) cs
            loop (String s :: acc) cs'
        | '-' :: d ::cs when Char.IsDigit(d) ->
            let n, cs = readNumber (Some '-') (d :: cs)
```

```
            loop (n :: acc) cs
        | '+' :: d ::cs when Char.IsDigit(d) ->
            let n, cs = readNumber (Some '+') (d :: cs)
            loop (n :: acc) cs
        | '#' :: 't' :: cs | '#' :: 'T' :: cs ->
            loop (Bool true :: acc) cs
        | '#' :: 'f' :: cs | '#' :: 'F' :: cs ->
            loop (Bool false :: acc) cs
        | '#' :: '\\' :: cs ->
            let c, cs' = character cs
            loop (c :: acc) cs'
        | '.' :: d :: _ when Char.IsDigit(d) ->
            let n, cs = readNumber None cs
            loop (n :: acc) cs
        | '.' :: cs -> loop (Dot :: acc) cs
        | '\'' :: cs -> loop (Quote :: acc) cs
        | ',' :: cs -> loop (Unquote :: acc) cs
        | d :: _ when Char.IsDigit(d) ->
            let n, cs = readNumber None cs
            loop (n :: acc) cs
        | [] -> acc
        | cs ->
            let s, cs' = symbol "" cs
            loop (Symbol s :: acc) cs'

    loop [] (Seq.toList source) |> List.rev
```

А.2 Вихідний код синтаксичного аналізатору

```
type SExpr =
    | Cons of SExpr * SExpr
    | Nil
    | Number of int
    | FloatNumber of float
    | String of string
    | Char of char
    | Symbol of string
    | Bool of bool
    | VoidValue
and Frame = Map<string, SExpr ref> ref

let (|List|ListImproper|ListWrong|) (e: SExpr) =
    let rec loop acc = function
        | Cons (x, (Cons(_) as y)) ->
            loop (x :: acc) y
        | Cons (x, Nil) ->
            let lst = List.rev (x :: acc)
            Choice1Of3(lst)
        | Cons (x, y) ->
            let lst = List.rev (y :: x :: acc)
            Choice2Of3(lst)
        | Nil -> Choice1Of3([])
        | _ -> Choice3Of3(())
    let retu = loop [] e
    retu

/// **Description**
///   Converts expession list in consed list : [1;2;3] -
> Cons(1, Cons(2, Cons(3, Nil)))
/// **Parameters**
///   * `exprs` - parameter of type `SExpr list`
///
/// **Output Type**
///   * `SExpr`
///
/// **Exceptions**
///
let exprsToList exprs =
    List.foldBack (fun x y -> Cons (x, y)) exprs Nil

/// **Description**
```

```
///   Converts expression list in conses, included dotted conses : [1;2;3] -
> Cons(1, Cons(2, 3))
/// **Parameters**
///   * `_arg1` - parameter of type `SExpr list`
///
/// **Output Type**
///   * `SExpr`
///
/// **Exceptions**
///
let rec exprsToDottedList = function
    | [Nil] -> Nil
    | x :: [y] -> Cons (x, y)
    | x :: xs -> Cons (x, exprsToDottedList xs)
    | [] -> failwith "Not expected for buildConses"

/// **Description**
///   Converts conses to expression list. Cons(1, Cons(2, Nil)) -> [1;2]
/// **Parameters**
///   * `_arg1` - parameter of type `SExpr`
///
/// **Output Type**
///   * `SExpr list`
///
/// **Exceptions**
///
let rec consToList = function
    | Cons (x, y) ->
        x :: (consToList y)
    | Nil -> []
    | e -> [e]

let rec consToListMap f = function
    | Cons (x, y) ->
        (f x) :: (consToListMap f y)
    | Nil -> []
    | _ -> failwith "Cons or Nil is expected"

let rec parse ts : SExpr =
    let rec parseList o (acc:SExpr list) ts =
        match ts with
        | Dot :: ts ->
            let elem, ts1 = parseExpr ts
            match ts1 with
            | Close :: ts2 -> exprsToDottedList (List.rev (elem :: acc)), ts2
```

```
            | wha -> failwithf "Close paren is expected after dot: %A" wha
        | Close :: ts ->
            if o = Open then
                exprsToDottedList (List.rev (Nil :: acc)), ts
            else failwithf "Close paren expected, got %A\n\n%A" o ts
        | Open :: ts ->
            let subList, ts1 = parseList Open [] ts
            parseList o (subList :: acc) ts1
        | CloseBr :: ts ->
            if o = OpenBr then
                exprsToDottedList (List.rev (Nil :: acc)), ts
            else failwith "Close bracket expected"
        | OpenBr :: ts ->
            let subList, ts1 = parseList OpenBr [] ts
            parseList o (subList :: acc) ts1
        | ts ->
            let elem, ts1 = parseNonlist ts
            parseList o (elem :: acc) ts1

    and parseNonlist ts =
        match ts with
        | Token.Number n :: ts ->
            (Number (int n)), ts
        | Token.FloatNum n :: ts ->
            (FloatNumber (float n)), ts
        | Token.Symbol n :: ts ->
            (Symbol (n.ToLower())), ts
        | Token.String n :: ts ->
            (String n), ts
        | Token.Bool n :: ts ->
            (Bool n), ts
        | Token.Char c :: ts ->
            (Char c), ts
        | Quote :: ts ->
            let expr, ts1 = parseExpr ts
            Cons (Symbol "quote", Cons (expr, Nil)), ts1
        | Unquote :: ts ->
            let expr, ts1 = parseExpr ts
            Cons (Symbol "unquote", Cons (expr, Nil)), ts1
        | ts -> failwithf "Expected a list element. %A" ts

    and parseExpr ts =
        match ts with
        | Open :: ts1 ->
            parseList Open [] ts1
```

```
        | OpenBr :: ts1 ->
            parseList OpenBr [] ts1
        | _ -> parseNonlist ts

    parseExpr ts |> fst

let rec sexprToString expr =
    let s = StringBuilder()
    let add (str:string) = s.Append(str) |> ignore
    let rec loop = function
        | Number n -> n.ToString() |> add
        | FloatNumber n -> n.ToString() |> add
        | String str ->
            add "\""
            str.Replace("\"", "\\\"") |> add
            add "\""
        | Symbol sy -> add sy
        | Bool true -> add "#t"
        | Bool false -> add "#f"
        | VoidValue -> add "#!void"
        | Char c ->
            add "#\\"
            add <| c.ToString()
        | Cons _ as c ->
            add "("
            processCons c
            add ")"
        | Nil -> add "()"
    and processCons = function
        | Cons (x, y) ->
            match y with
            | Cons _ ->
                loop x
                add " "
                processCons y
            | Nil ->
                loop x
            | _ ->
                loop x
                add " . "
                loop y
        | _ -> failwith "Cons is expected"
    and printLambdaMacro name args body=
        let args = List.map Symbol args
        [ yield Symbol name
```

```
            yield! args
            yield body ]
        |> exprsToList
        |> sexprToString
        |> add
    loop expr

    s.ToString()

let sexprToCompactString sexpr =
    let s = StringBuilder()

    let newline() = s.AppendLine() |> ignore
    let indent n = s.Append(' ', n).Append("- ") |> ignore

    let rec loop sexpr n =
        match sexpr with
        | Cons(x, y) ->
            s.Append("Cons") |> ignore
            newline()
            indent n
            loop x (n + 1)
            newline()
            indent n
            loop y (n + 1)
        | _ ->
            s.Append(sprintf "%A" sexpr) |> ignore

    loop sexpr 0
    s.ToString()

let stringToSExpr (s : string) = s |> tokenize |> parse

let symbolsToStrings sexprs =
    List.map (function
        | Symbol name -> name
        | _ -> failwith "symbolsToStrings: Symbol expected") sexprs
```

А.3 Вихідний код програми порівняння інваріантів

```
(define (merge-sort pred ls)
  (define (split ls)
    (letrec ([split-h (lambda (ls ls1 ls2)
                        (cond
                          [(or (null? ls) (null? (cdr ls)))
                          (cons (reverse ls2) ls1)]
                          [else (split-h (cddr ls)
                                  (cdr ls1) (cons (car ls1) ls2))]))])
      (split-h ls ls '())))
  (define (merge pred ls1 ls2)
    (cond
      [(null? ls1) ls2]
      [(null? ls2) ls1]
      [(pred (car ls1) (car ls2))
        (cons (car ls1) (merge pred (cdr ls1) ls2))]
      [else (cons (car ls2) (merge pred ls1 (cdr ls2)))]))
  (cond
    [(null? ls) ls]
    [(null? (cdr ls)) ls]
    [else (let ([splits (split ls)])
            (merge pred
              (merge-sort pred (car splits))
              (merge-sort pred (cdr splits))))]))

;;; Reads input port ip.
;;; Returns a list of lines. Each line is a list of chars.
(define (split-lines ip)
  (let loop ([line '()] [lines '()])
    (let ([c (read-char ip)])
      (if (eof-object? c)
        (reverse (if (empty? line) lines (cons (reverse line) lines)))
        (if (eq? c #\newline)
          (loop '() (cons (reverse line) lines))
          (loop (cons c line) lines))))))

;;; Converts a line of characters into a list of names.
;;; Names are separated by space.
(define (line->names line)
  (define (add-name name names)
    (cons (list->string (reverse name)) names))
  (let loop ([chars line] [name '()] [names '()])
    (cond
```

```
      [(empty? chars)
        (if (empty? name)
          names
          (add-name name names))]
      [(eq? (car chars) #\space)
        (if (empty? name)
          (loop (cdr chars) name names)
          (loop (cdr chars) '()
                (add-name name names)))]
      [else (loop (cdr chars) (cons (car chars) name) names)])
  ))

;;; Extracts invariants from tina-generated file.
;;; Receives lines.
;;; Finds a line with word "invariant", skips a line after that.
;;; Skips everything after the first empty line.
;;; Traverses each line and removes everything after the first parentheses.
(define (extract-from-tina lines)
  (define (remove-paren line)
    (let loop ([acc '()] [chars line])
      (cond
        [(empty? chars) (reverse acc)]
        [(eq? (car chars) #\( ) (reverse acc)]
        [else (loop (cons (car chars) acc) (cdr chars))])))

  (let* ([part1
          (let loop ([lines lines])
            (if (equal? (car lines) '(#\i #\n #\v #\a #\r #\i #\a #\n #\t))
              (cdr (cdr lines))
              (loop (cdr lines))))]
         [part2
          (let loop ([acc '()] [lines part1])
            (if (empty? (car lines))
              (reverse acc)
              (loop (cons (car lines) acc) (cdr lines))))]
          [part3 (map remove-paren part2)])
    part3))

;;; Compare two invariants. Each invariant is a list of names.
(define (invariant<? inv1 inv2)
  (let loop ([inv1 inv1] [inv2 inv2])
    (cond
      [(and (empty? inv1) (empty? inv2)) #t]
      [(empty? inv1) #t]
      [(empty? inv2) #f]
```

```
      [(string=? (car inv1) (car inv2)) (loop (cdr inv1) (cdr inv2))]
      [(string<? (car inv1) (car inv2)) (<= (length inv1) (length inv2))]
      [else (< (length inv1) (length inv2))])))

(define (sort-invariants invs)
  (let* ([invs (map (lambda (inv) (merge-sort string<? inv)) invs)]
         [invs (merge-sort invariant<? invs)])
    invs))

(define (read-invariants filename tina?)
  (let* ([ip (open-input-file filename)]
         [lines (split-lines ip)]
         [lines (if tina? (extract-from-tina lines) lines)]
         [name-lines (map line->names lines)]
         [invariants (sort-invariants name-lines)])
    invariants))

(define (invariants-equal? invs1 invs2)
  (cond
    [(and (empty? invs1) (empty? invs2)) #t]
    [(empty? invs1) #f]
    [(empty? invs2) #f]
    [(string=? (car invs1) (car invs2))
      (invariants-equal? (cdr invs1) (cdr invs2))]
    [else #f]))

(define (compare-invariants file1 tina1? file2 tina2?)
  (let* ([invs1 (read-invariants file1 tina1?)]
         [invs2 (read-invariants file2 tina2?)])
    (display (if (equal? invs1 invs2)
                 "Invariants are equal"
                 "Invariants are not equal"))
    (newline)))

(compare-invariants "misc/file1.txt" #f "misc/file2.txt" #t)

(exit-scheme)
```