DOI 10.34185/1562-9945-5-142-2022-09
УДК 004.4'412

І.О. Сокол, О.С. Волковський

ЛЕКСИЧНИЙ АНАЛІЗ ПРОГРАМНОГО КОДУ

Анотація. Будь-який розбір програмного коду починається з лексичного аналізу. Попри те, що лексичний аналіз вважається відносно простим етапом, він грає ключову роль у всій системі аналізу та перетворення програмного коду, а також має велику кількість як теоретичних так і практичних особливостей, що потребують ретельного дослідження. В даній статті розглядаються визначення лексичного аналізатора, принципи його побудови, теоретичні особливості та особливості практичної реалізації.
Ключові слова: лексичний аналізатор, лексема, токен, регулярні вирази, теорія формальних мов, недетермінований кінцевий автомат, детермінований кінцевий автомат.

Етапи перекладу програмного коду, які наведені на рисунку 1, можна розділити на дві глобальні фази – фаза аналізу та фаза синтезу. Етап лексичного аналізу поряд з синтаксичним та семантичним аналізом належить до фази аналізу [1].

Рисунок 1 – Етапи перекладу програмного коду

Лексичний аналіз – це процес перетворення послідовності символів вихідного програмного коду в послідовність токенів (груп символів, що відповідають певним шаблонам) та визначення їх типів. Програма чи функція, що виконує лексичний наліз, називається лексичним аналізатором, токенізатором чи сканером [2].

Лексичний аналізатор сканує вихідний код рядок за рядком, приймаючи на вхід лексеми він генерує потік токенів. Наприклад, в компіляторах, часто, лексичний аналізатор є функцією, яку використовує синтаксичний аналізатор для отримання чергового токену.

Лексеми аналогічні словам з природніх мов з однією невеликою різницею: слова окремо мають власні значення, тоді як лексеми частіше всього передають зміст групами. Наприклад, визначимо наступний рядок коду:

$$x = a + b * c; \quad (1)$$

У виразі (1) «x» окремо не передає жодного значення, доки ми не розглянемо весь арифметичний вираз. Токен є значенням лексеми. Тоді з точки зору значення у даному виразі «x» є ідентифікатором, символ «=» є оператором (оператором присвоєння), «a» - ідентифікатор і т.д. Як бачимо лексеми «x», «a», «b», «c» мають різний вигляд, але значення передають одне і те саме. А отже, роботою лексичного аналізатора є знаходження значення кожної лексеми. Загалом в мовах програмування виділяють наступні типи токенів [3]:

- ідентифікатор;
- оператор;
- константа;
- ключове слово;
- літерал;
- знак пунктуації;
- спеціальний символ.

На рисунку 2 наведений функціональний склад лексичного аналізатора. Лексеми передаються до фази сканування, яка в свою чергу усуває елементи, що не є токенами, такі як коментарі, послідовні пробіли та інше. В результаті, під час фази аналізу формуються токени. Звичайно, якщо лексичному аналізатору не вдалося знайти відповідний токен до лексеми, він повинен сформувати повідомлення про помилку з необхідною інформацією.

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

Рисунок 2 – Функціональний склад лексичного аналізатора

Процес формування токенів потребує певний набір шаблонів для знаходження значень лексем. Стандартним методом опису шаблонів пошуку є регулярні вирази.

Регулярні вирази (англ. regular expressions) – формальна мова, яка використовується в комп'ютерних програмах, що працюють з текстом, для пошуку та здійснення маніпуляцій з підрядками у тексті, заснована на використанні метасимволів. Для пошуку використовується рядок-зразок, який складається з символів та метасимволів та задає правило пошуку [4].

Регулярні вирази мають множину діалектів, особливу популярність набули наступні діалекти: POSIX, Perl, PCRE, ECMAScript, Python. В теорії формальних мов регулярні вирази складаються з констант та операторів, які визначають множину рядків та множину операцій над ними відповідно. На цьому кінцевому алфавіті ∑ визначені наступні константи [5]:

- порожня множина $\boldsymbol{\emptyset}$ позначає $\boldsymbol{\emptyset}$;
- порожній рядок ε позначає множину $\{\varepsilon\}$;
- рядок $\boldsymbol{a}$ в ∑ позначає множину $\{\boldsymbol{a}\}$;

та наступні операції [5]:

- конкатенація (concatenation) $\boldsymbol{RS}$ позначає набір рядків, який можна отримати конкатенацією множин визначених R та S (у цьому порядку), наприклад: нехай R позначає {“ab”, “c”} та S позначає {“d”, “ef”}, тоді RS позначає {“abd”, “abef”, “cd”, “cef”};

- чергування (alternation) $\boldsymbol{R|S}$ позначає об'єднання множин, описаних R та S, наприклад: якщо R описує {“ab”, “c”} та S описує {“ab”, “d”, “ef”}, тоді вираз $R|S$ описує {“ab”, “c”, “d”, “ef”};

- замикання Кліні (Kleene closure) $\boldsymbol{R^*}$ позначає найменшу надмножину множин, описаної R, що містить ε і є замкнутою відносно конкатенації рядків, це набір всіх рядків, який може бути створений завдяки конкатенації будь-

якого кінцевого числа (включаючи нуль) рядків із набору описаного *R*, наприклад: якщо *R* позначає {“0”, “1”}, тоді *R** позначає набір всіх кінцевих двійкових рядків (включаючи порожній рядок).

Для прикладу створимо набір шаблонів на основі регулярних виразів для наступної лексики мови програмування:

1. Мова програмування має одне ключове слово “while”.

2. Ідентифікатор може складатися з літер латинського алфавіту та цифр, але повинен починатися з літери.

3. Ціле число може бути 0 або складатися з цифр, за умови того, що перша цифра не буде 0. Ціле число може містити знак на початку.

4. Дійсне число з фіксованою крапкою (decimal), наприклад 134.0093, -76.11.

5. Дійсне число з плаваючою крапкою (real), наприклад 1.3400E+02, -98.345E-5, 156E8.

Для зручності можна сформувати наступні позначення, використовуючи описані операції над рядками:

1. $P^+ = PP^*$. Нехай *P* це рядок “a”, тоді P^+ буде визначати наступну множину: {“a”, “aa”, “aaa”, “aaaa”, ...}.

2. *P?* = *P*| ε. Нехай *P* це рядок “a”, тоді *P*? буде визначати наступну множину: {“a”, “ε”}.

3. *[a-z]* = *a*|*b*|*c*|...|*z*. Позначення буде визначати множину {“a”, “b”, “c”, “d”, ..., “z”}.

Також, для зручності можна давати імена регулярним виразам, щоб потім посилатися на них за іменем. В результаті, набір регулярних виразів для наведеної лексики буде наступним:

```
while-keyword = while
letter = [a-zA-Z]
digit = [0-9]
identifier = letter (letter | digit)*
sign = + | -
integer = sign? (0 | [1-9] digit*)
decimal = integer . digit*
real = (integer | decimal) E sign? digit+
```

Існує два основних правила під час визначення токену [6]:

1. Якщо є неоднозначність, коли у вхідних даних є збіг з декількома шаблонами різних розмірів, то обирається шаблон з найбільшою довжиною – правило найдовшого збігу. Наприклад, якщо в коді зустрінеться послідовність си-

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

мволів “while8”, то згідно сформованих регулярних виразів ми отримаємо токен ідентифікатора, а не токен ключового слова while, тому що шаблон ключового слова має менше символів.

2. Якщо є шаблони, які відповідають однаковій максимальній кількості символі, то обирається шаблон, який вказаний першим – правило пріоритету. Наприклад, якщо в коді зустрінеться послідовність символів “while”, то дана лексема відповідає одночасно токену ключового слова while і токену ідентифікатора, проте, так як шаблон ключового слова описаний раніше в наборі шаблонів, то буде сформований саме токен ключового слова.

На практиці лексичні аналізатори використовують дещо інший механізм пошуку значень лексем. А саме кінцеві автомати (finite-state machine) – особливий різновид автомату - абстракції, що використовується для опису шляху зміни стану об'єкта в залежності від поточного стану та інформації отриманої ззовні. Його особливістю є скінченність множини станів автомату [7].

В реалізаціях лексичних аналізаторів застосовують різновид кінцевого автомату, а саме детермінований кінцевий автомат. Це обумовлено практичною ефективністю використання детермінованого алгоритму на реальних обчислювальних машинах. Існують інструменти, які допомагають переводити відносно легкі у сприйнятті людиною регулярні вирази в кінцеві автомати. Наприклад, таким інструментом є GNU Flex. GNU Flex дозволяє описати лексичний аналізатор простою мовою Flex, перерахувавши регулярні вирази для окремих токенів і вказавши код, який буде виконуватись під час зіставлення кожного токену. В результаті Flex генерує код мовою програмування C або C++. Отриманий код використовує детермінований кінцевий автомат для аналізу тексту [8].

Розглянемо побудову кінцевих автоматів на прикладі декількох токенів з мови програмування C, а саме ключове слово if, ідентифікатор та ціле число. На рисунку 3 наведені індивідуальні кінцеві автомати для цих токенів з власними початковими станами. Для спрощення сприйняття в рисунках кінцевих автоматів спеціально не будуть зображуватись невизначені стани, які приводять до помилки визначення токену. Адже зрозуміло, що за відсутності переходу, який відповідає вхідним даним ми будемо формувати помилку.

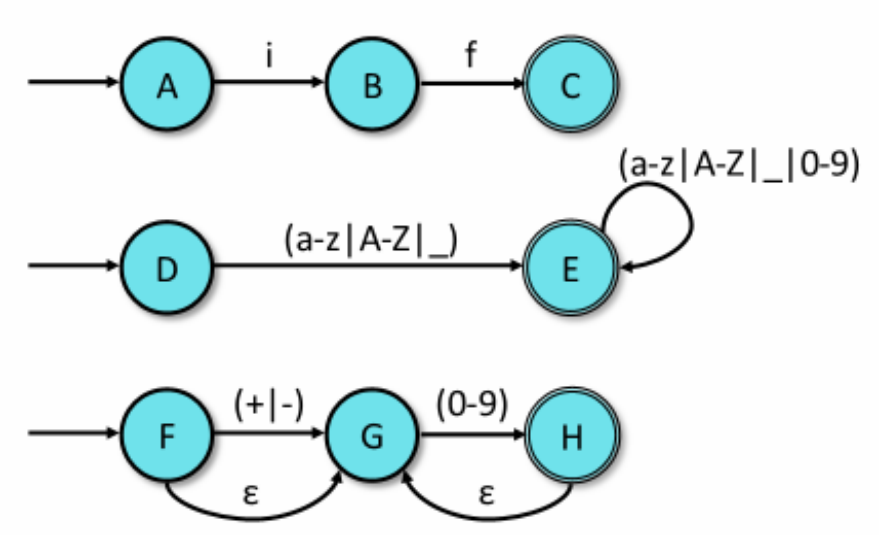

Рисунок 3 – Індивідуальні кінцеві автомати для токенів

Отже, розглянемо автомати на рисунку 3:

1. Для ключового слова if. З початкового стану *A*, якщо ми отримуємо літеру “i”, то ми переходимо до наступного стану *B*. Потім з *B* отримавши літеру “f” ми закінчуємо у кінцевому стані *C*.

2. Для ідентифікатора. З початкового стану *D*, отриманий символ повинен бути будь-якою маленькою літерою з діапазону латинського алфавіту від “a” до “z” або будь-якою великою літерою з того ж діапазону, або нижнім підкресленням, тоді ми переходимо до фінального стану *E*, тому що одна літера також може бути ідентифікатором. Додатково ідентифікатор може мати будь-яку кількість літер, нижніх підкреслень або цифр, але вони повинні слідувати за будь-якою літерою чи нижнім підкресленням, а тому ми робимо перехід з кінцевого стану *E* в нього ж, доки не завершиться процес переходу.

3. Для цілого числа. З початкового стану *F*, якщо ми отримуємо будь-який знак (позитивний чи негативний), то ми переходимо до стану *G*, і звідси, якщо наступний символ будь-який з діапазону цифр від 0 до 9 ми потрапляємо до кінцевого стану *H*. Цього достатньо для цілого числа з однією цифрою. Для двох або більше цифр в числі ми маємо ε-перехід від кінцевого стану *H* в *G*. Ɛ-перехід – це перехід без використання вхідного символу, зручний спосіб моделювання. Тепер, якщо ми не бачимо іншого переходу, ми можемо перемістися в стан *G*, і тоді можемо мати будь-яку кількість цифр. Між іншим, ціле число може не мати жодного знаку на початку, тому що для позитивного числа позитивний знак ставити не обов’язково. Тому необхідно сформувати ε-перехід від початкового стану *F* в стан *G*.

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

Під час фази аналізу в лексичному аналізаторі не застосовуються відокремленні кінцеві автомати, використовується один кінцевий автомат, що об'єднує усі відокремленні автомати. Отже, нам необхідно виконати об'єднання кінцевих автоматів. В нашому випадку можна ввести новий початковий стан S і побудувати ε-переходи до існуючих автоматів (рисунок 4). В результаті сформувався недетермінований кінцевий автомат (NFA). Точніше це ε-NFA – різновид недетермінованого кінцевого автомата з ε-переходами [3].

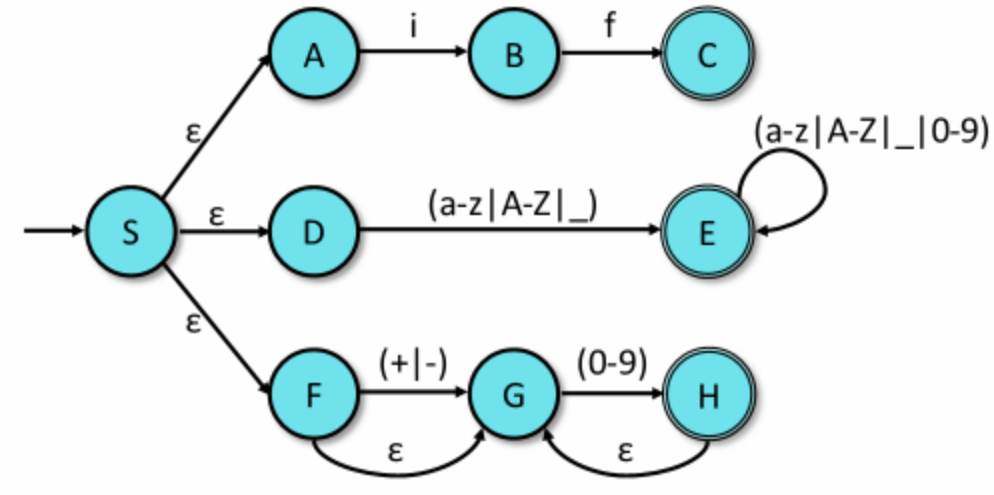

Рисунок 4 – Недетермінований кінцевий автомат з ε-переходами

В недетермінованих автоматах вхід може привести до одного, більше ніж одного або зовсім без переходу для даного стану. Якщо в теорії дана особливість не має критичного значення, то на практиці недетермінований автомат не підходить для впровадження в машинних алгоритмах. Лексичні аналізатори використовують детерміновані кінцеві автомати (DFA). На відміну від недетермінованих, в детермінованих автоматах кожен стан має лише один перехід для кожного входу. При цьому, існує алгоритм трансформації будь-якого NFA в складніший DFA з однаковою функціональністю [7]. Отже, на рисунку 5 наведений еквівалентний до автомата з рисунку 4 детермінований кінцевий автомат.

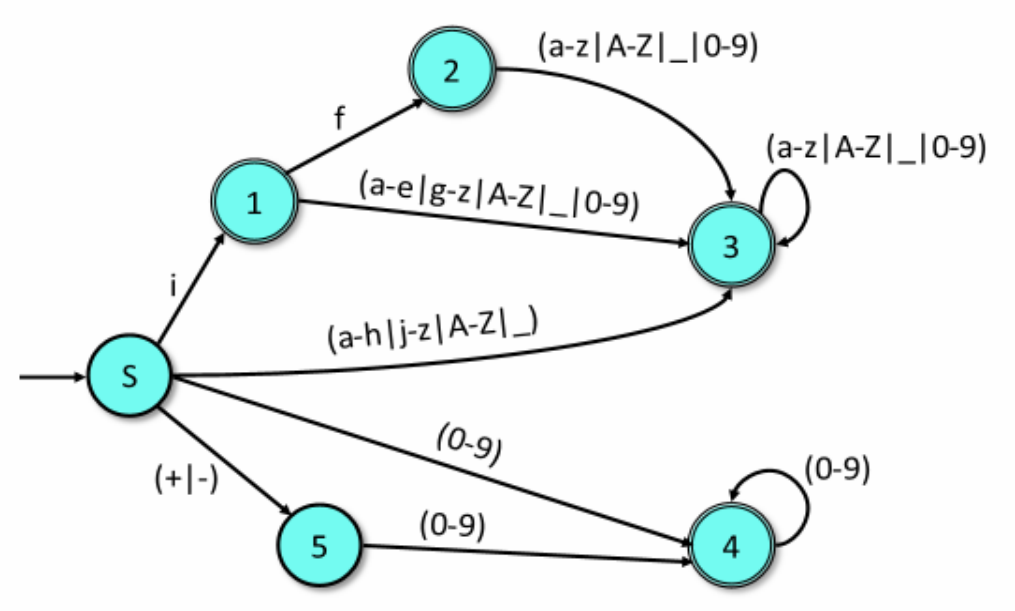

Рисунок 5 – Детермінований кінцевий автомат

Він розпізнає ключове слово if, ідентифікатор та ціле число. Розглянемо детальніше:

Починаючи зі стартового стану *S*, якщо автомат отримує маленьку літеру “i”, ми переходимо до стану *1*. Звідси, якщо ми отримуємо маленьку літеру “f”, то переходимо до стану *2*. Так як стан *2* є кінцевим станом, то якщо ми зупиняємося тут, це означає, що автомат визначив вхідну лексему як токен ключового слова if.

Якщо вхідна лексема має більше символів, ніж “if”, наприклад, “iffy”, то ми перейдемо до кінцевого стану *3*, і далі будемо використовувати перехід з *3* стану в нього ж. Якщо ми зупинимось на стані *3*, це означає, що автомат визначив вхідну лексему як токен ідентифікатора. На рисунку 5 видно, що стан *1* також є кінцевим станом, адже, якщо ми отримаємо літеру “i” від початкового стану *S* і за нею не буде слідувати літера “f”, тоді це не ключове слово if, а ідентифікатор. Якщо ми зупинимося в стані *1*, то автомат визначить лексему як ідентифікатор. Тепер розглянемо перехід від стану *1* до стану *3*. Тут спеціально перевіряється діапазон маленьких літер від “a” до “e” і від “g” до “z”, тобто ми виключили з діапазону літеру “f”, адже літера “f” є умовою переходу від стану *1* до стану *2*. За такого діапазону ми просуваємося до стану *3* і можемо визначати лексему як ідентифікатор. Так само працює перехід від стану *S* до стану *3*, якщо ми отримаємо будь-яку літеру (або нижнє підкреслення) окрім маленької літери “i”, то ми перейдемо до стану *3*.

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

Очевидно, що перехід від стану *S* до стану *4* дозволяє визначити вхідну лексему як токен цілого числа. Перехід від S до стану *5* враховує цілі числа зі знаком на початку.

В результаті кінцевий стан *2* визначає вхідну лексему як токен ключового слова if, кінцеві стани *1* та *3* – як токен ідентифікатора, кінцевий стан *4* – як токен цілого числа.

Детермінований кінцевий автомат як такий, що наведений на рисунку 5 реалізується у фазі аналізу (рисунок 2) в лексичному аналізаторі. Йому на вхід передаються лексеми, за виключенням елементів, що не є токенами, для визначення значень цих лексем, тобто для формування токенів.

Отже, ми розглянули загальне визначення лексичного аналізатора та принципи його побудови, ретельно дослідили теоретичне підґрунтя лексичного аналізу, а також особливості практичної реалізації на обчислювальних машинах.

ЛІТЕРАТУРА

1. В.Г. Павлов, К.О. Шапран Основні принципи побудови лексичних аналізаторів. Матеріали IV Міжнародної науково-технічної конференції молодих учених та студентів. Актуальні задачі сучасних технологій – Тернопіль 25-26 листопада 2015. С. 68-69.
2. Lexical analysis [Електронний ресурс] – Режим доступу: https://en.wikipedia.org/wiki/Lexical_analysis
3. Introduction to compiler design (PPT) [Електронний ресурс] – Режим доступу: https://www.nesoacademy.org/cs/12-compiler-design/ppts/01-introduction-to-compiler-design
4. Регулярные выражения [Електронний ресурс] – Режим доступу: https://bit.ly/3UrPRvp
5. Regular expression [Електронний ресурс] – Режим доступу: https://en.wikipedia.org/wiki/Regular_expression
6. Regular Expressions (REs) [Електронний ресурс] – Режим доступу: https://lambda.uta.edu/cse5317/notes/node7.html
7. Скінченний автомат [Електронний ресурс] – Режим доступу: https://bit.ly/3ukLJT4
8. Изучаем генератор кода GNU Flex [Електронний ресурс] – Режим доступу: https://ps-group.github.io/compilers/gnu_flex_doc

REFERENCES

1. V.G. Pavlov, K.O. Shapran Basic principles of building lexical analyzers. Materials of the IV International Scientific and Technical Conference of Young Scientists and Students. Actual tasks of modern technologies - Ternopil, November 25-26, 2015. P. 68-69.
2. Lexical analysis [Electronic resource] – Resource access mode: https://en.wikipedia.org/wiki/Lexical_analysis
3. Introduction to compiler design (PPT) [Electronic resource] – Resource access mode: https://www.nesoacademy.org/cs/12-compiler-design/ppts/01-introduction-to-compiler-design
4. Regular expressions [Electronic resource] – Resource access mode: https://bit.ly/3UrPRvp
5. Regular expression [Electronic resource] – Resource access mode: https://en.wikipedia.org/wiki/Regular_expression
6. Regular Expressions (REs) [Electronic resource] – Resource access mode: https://lambda.uta.edu/cse5317/notes/node7.html
7. Finite state machine [Electronic resource] – Resource access mode: https://bit.ly/3ukLJT4
8. Learning the GNU Flex Code Generator [Electronic resource] – Resource access mode: https://ps-group.github.io/compilers/gnu_flex_doc

Received 12.09.2022.
Accepted 16.09.2022.

Lexical analysis of program code

The growing volume of technologies, the end of actively used development tools support, outdated API etc., entails the need of program codes conversion. In IT companies and not only, often begged the question of deprecated software support, which customers continue to use, or translation of current software to actual technologies. It is more rational for programmers to use the conversion and save most of code base, than rewriting all software by hand, even if manual adjustment is needed. At this moment, there are few high-quality code conversion systems. Largely, conversion systems work well only with similar programming languages. The task of program codes conversion is a deep and complex topic. To convert the software code, you must first analyze, select components and form a structural representation. Any analysis of program code begins with lexical analysis. Although lexical analysis is considered a relatively simple step, it plays a key role in the entire system of analysis and transformation of software code, and also has a large number of both theoretical and practical features that require careful study.

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

This article considers the definition of the lexical analyzer, its functional composition and principles of construction, provides key differences between the lexeme and the token. Two approaches have been proposed and considered to solve the search for tokens in the program code: regular expression search and finite state machine search. For these approaches, examples of the formation of search templates under certain rules of vocabulary were given. As a result, the optimality of the use of deterministic finite state machines during the practical implementation of the lexical analyzer on real computing machines was substantiated.

Сокол Іван Олександрович – аспірант, кафедра комп'ютерних наук та інформаційних технологій, Дніпровський національний університет ім.О. Гончара.
Волковський Олег Степанович – к.т.н., доцент кафедри КНІТ, Дніпровський національний університет ім.О. Гончара

Sokol Ivan – graduate student, Department of computer science and information technologies, Oles Honchar Dnipro National University.
Volkovskyi Oleg – PhD in Technical Science, associate professor of CSIT Department, Oles Honchar Dnipro National University.