МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Київський національний університет імені Тараса Шевченка
Факультет кібернетики

МЕТОДТЧНІ РЕКОМЕНДАЦІЇ
для лабораторного практикуму побудови мовних процесорів з курсу «Системне програмування»

Київ-2017

Зміст

1. Мови програмування та мовні процесори

При вивченні мов програмування, як правило, виділяють три аспекти:

- Прагматичний;
- Семантичний;
- Синтаксичний.

Прагматичний аспект (прагматика мови програмування) визначає клас задач, на рішення яких орієнтується мова програмування. Як правило, прагматичний аспект менш формалізований в порівнянні з семантичним та синтаксичним аспектами.
З урахуванням на рішення задач певного класу мови програмування можна поділити на процедурні та непроцедурні.

Процедурні мови програмування орієнтовані перш за все на опис (визначення) алгоритмів, тобто по суті використовуються для побудови процедур обробки даних. До таких мов ми відносимо всім відомі мови програмування, такі як Pascal, Fortran, C та ін.

Непроцедурні мови програмування на відміну від процедурних неявно визначають процедури обробки даних. Частіше всього такі мови використовуються для побудови завдань на обробку даних. При цьому, при допомозі інструкцій непроцедурної мови програмування визначається що необхідно зробити з даними і явно не визначається як (при допомозі яких алгоритмів) необхідно розв'язати задачу. До непроцедурних мов програмування ми відносимо командні мови операційних систем, мови управління в пакетах прикладних програм та ін.

Як процедурні, так і непроцедурні мови програмування можуть орієнтуватися як на декілька класів задач, так і конкретну предметну область. В першому випадку ми будемо говорити про універсальні мови програмування (Pascal, Fortran, C), в другому – про спеціалізовані мови програмування (Snobol, Lisp).

Семантичний аспект (семантика мови програмування) визначається шляхом конкретизації базових функцій обробки даних, набору конструкцій управління та методами побудови більш "складних" програм на основі "простих".

Наприклад, визначивши як базовий тип даних "рядок" ми повинні запропонувати "традиційний" набір функцій обробки таких даних: порівняння рядків, виділення частини рядка, конкатенацію рядків та ін.

Семантика мови програмування має бути визначена формально, інакше в подальшому неможливо буде побудувати відповідний мовний процесор. На сьогодні існують два основних напрямки визначення семантики мов програмування: методи денотаційної семантики та методи операційної семантик. Методи денотаційної семантики базуються на відповідних алгебрах, методи операційної семантики базуються на синтаксичних структурах програм.

Синтаксичний аспект (синтаксис мови програмування) визначає набір синтаксичних конструкцій мови програмування, які використовуються для нотації (запису) семантичних одиниць в програмі. Про синтаксис мови програмування можна сказати як про форму, яка є суть похідною від семантики. Для визначення (опису) синтаксису мови програмування використовуються як механізми, що орієнтовані на синтез, так і механізми, орієнтовані на аналіз. Задачі аналізу та синтезу синтаксичних структур програм – це дуальні задачі. Їх конкретизацію ми будемо розглядати в наступних розділах.

Виходячи з вищенаведеного, щоб побудувати мову програмування потрібно:

- визначити клас (класи) задач, на розв'язок яких орієнтована мова програмування;
- виділити базові типи даних та функції їх обробки, указати конструкції управління в програмах. Побудувати механізми конструювання більш складних програм та структур даних на основі більш простих одиниць;
- визначити синтаксис мови програмування.

Мовні процесори реалізують мови програмування. Точніше, мовний процесор призначений для обробки програм відповідної мови програмування. З точки зору прагматики, мовні процесори діляться на транслятори та інтерпретатори.

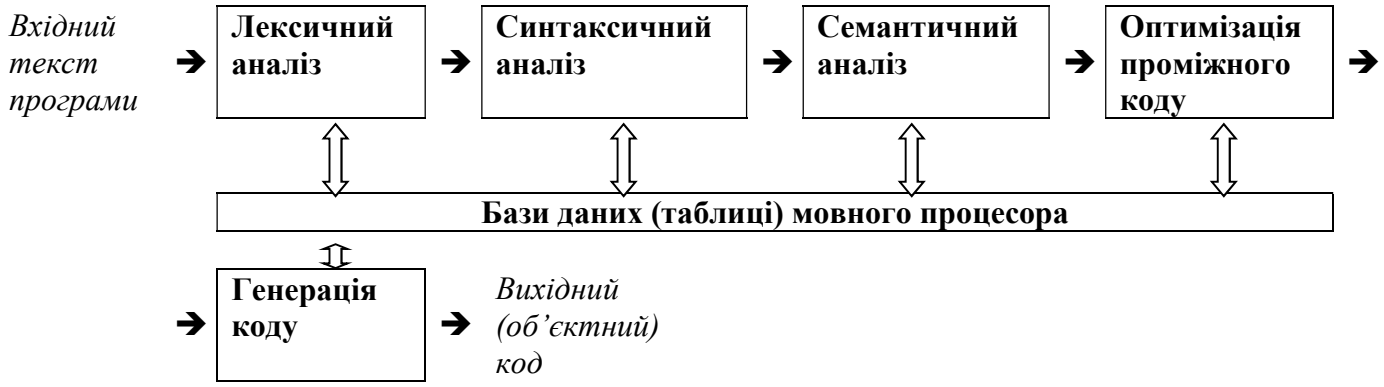
Мовний процесор типу транслятор (транслятор) – це програмний комплекс, котрий на вході отримує текст програми на вхідній мові, а на виході видає версію програми на вихідній мові, що називається об'єктною мовою. В більшості випадків як об'єктна мова виступає мова команд деякої обчислювальної машини. Серед трансляторів можна виділити дві програмні системи:

- компілятори – транслятори з мов програмування високого рівня;
- асемблери – транслятори машинно-орієнтованих мов програмування.

Мовний процесор типу інтерпретатор (інтерпретатор) – це програмний комплекс, котрий на вході отримує текст програми на вхідній мові та вхідні дані, які в подальшому обробляються програмою, а на виході видає результати обчислень (вихідні дані).

Оскільки транслятори та інтерпретатори реалізують мови програмування, вони мають спільні риси: їх структура досить схожа, в основу їх реалізації покладено спільні теоретичні результати та практичні методи реалізації.

Структура транслятора



Призначення основних компонентів транслятора:

1. *Лексичний аналізатор*.

Вхід: вхідний текст (послідовність літер) програми.

Вихід: послідовність лексем програми.

Лексема – це ланцюжок літер, що має певний зміст. Всі лексеми мови програмування (їх кількість, як правило, нескінчена) можна розбити на скінчену множину класів. Для більшості мов програмування актуальні наступні класи лексем:

- зарезервовані слова;
- ідентифікатори;
- числові константи (цілі та дійсні числа);
- літерні константи;
- рядкові константи;
- коди операцій;
- коментарі. Коментарі безпосередньо не несуть інформації щодо структури програми. В подальшому вони не використовуються, тобто не передаються синтаксичному аналізатору.
- дужки та інші елементи програми.

2. *Синтаксичний аналізатор*.
   Вхід: послідовність лексем програми.
   Вихід: - "Так" + синтаксична структура (синтаксичний терм) програми,
   - "Ні" + синтаксичні помилки в програмі.
3. *Семантичний аналізатор.*
   Вхід: Синтаксичний терм програми.
   Вихід: - "Так" + семантична структура (семантичний терм) програми,
   - "Ні" + семантичні помилки в програмі.
4. *Оптимізація проміжного коду.*
   Вхід: семантичний терм програми.
   Вихід: оптимізований семантичний терм програми.

Оптимізація – це еквівалентне перетворення програми на основі певних критеріїв. Серед критеріїв оптимізації можна виділити оптимізацію по пам'яті та оптимізацію по швидкості виконання результуючої програми. В залежності від підходів по оптимізації програми можна розглядати машиннозалежні та машиннонезалежні методи оптимізації. На відміну від машиннонезалежних методів машиннозалежні методи оптимізації враховують архітектурні особливості ЕОМ, наприклад, наявність апаратного стека, наявність вільних регістрів тощо.

5. *Генерація об'єктного коду.*
   Вхід: семантичний терм програми.
   Вихід: результуючий (об'єктний) код програми.

2. Лексичний аналіз в мовних процесорах

Призначення: перетворення вхідного тексту програми з формату зовнішнього представлення в машинноорієнтований формат – послідовність лексем.

Лексема – це ланцюжок літер елементарний об'єкт програми, що несе певний семантичний зміст. В подальшому кожну лексему будемо представляти як пару

(<клас_лексеми, ім'я_лексеми>)

В більшості мов програмування для визначення класів лексем достатньо скінчених автоматів.

2.1 Скінчені автомати

Означення: Недетермінований скінчений автомат – це п'ятірка

$M = < Q, \Sigma, \delta, q_0, F>$, де

- $Q = \{q_0, q_1, .. , q_{n-1}\}$ – скінчена множина станів автомата;
- $\Sigma = \{a_1, a_2, .., a_m\}$ – скінчена множина вхідних символів (вхідний алфавіт);
- $q_0 \in Q$ – початковий стан автомата;
- δ – відображення множини $Q^*\Sigma$ в множину $P(Q)$. Відображення δ як правило називають функцією переходів;
- $F \subseteq Q$ – множина заключних станів. Елементи з F називають заключними або фінальними станами.

Якщо M – скінчений автомат, то пара $(q, w) \in Q_*\Sigma^*$ називається конфігурацією автомата M. Оскільки скінчений автомат – це дискретний пристрій, він працює по тактам. Такт скінченого автомата M задається бінарним відношенням $|=$, яке визначається на конфігураціях:

$(q_1,aw) |= (q_2,w)$, якщо $\delta(q_1, a)$ містить q_2 та для всіх **$w \in \Sigma^*$.**

Означення. Скінчений автомат M розпізнає (допускає) ланцюжок w, якщо

$(q_0, w) |=^* (q, \varepsilon)$ для деякого **$q \in F$**, де

$|=^*$ - рефлексивно-транзитивне замикання бінарного відношення $|=$.

Означення. Мова, яку допускає автомат M (розпізнає автомат M)

$L(M)=\{ w \mid w \in \Sigma^*$ та $(q_0, w) |=^* (q, \varepsilon)$, $q \in F \}$

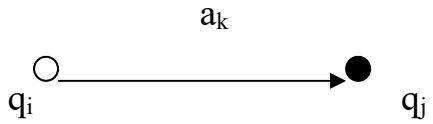
На практиці, при визначенні скінченого автомата M, використовують декілька способів визначення функції δ, наприклад:

- це табличне визначення δ;
- діаграма проходів скінченого автомата.

Табличне визначення функції δ - це таблиця $M(q_i,a_j)$, де $a_j \in \Sigma$, $q_i \in Q$, тобто

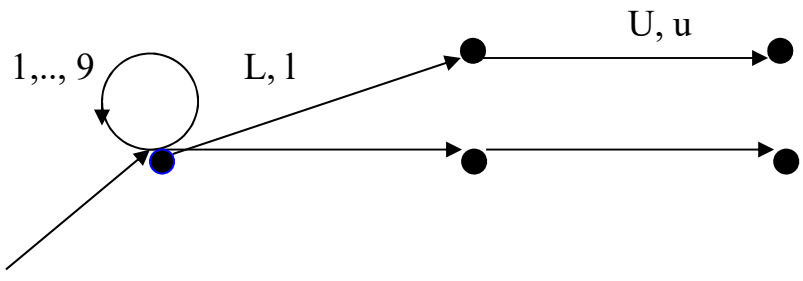
$$M(q_i,a_j) = \{ q_k \mid , q_k \in \delta(q_i,a_j) \}$$

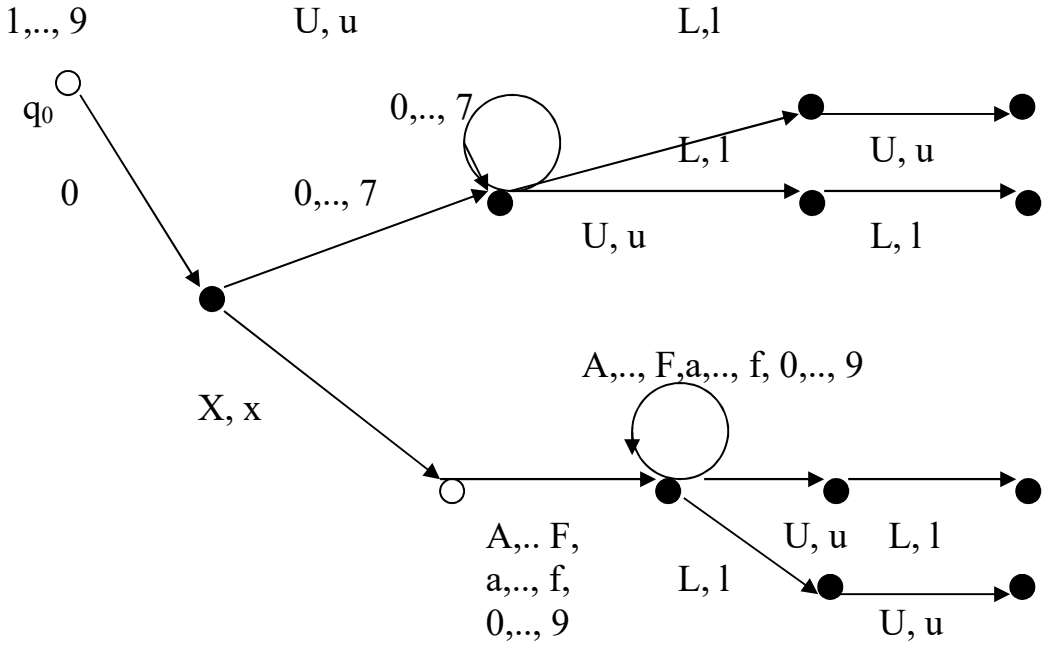
Діаграма переходів скінченого автомата M - це невпорядкований граф G(V, P), де V – множина вершин графа, а P – множина орієнтованих дуг, причому з вершини q_i у вершину q_j веде дуга позначена a_k, коли $q_j \in \delta(q_i,a_k)$. На діаграмі переходів скінченого автомата це позначається так:



В подальшому, на діаграмі переходів скінченого автомата M елементи з множини заключних станів будемо позначити так: ● q_i.

Приклад 1. Побудуємо діаграму переходів скінченого автомата M, який розпізнає множину цілочислових констант мови C.





Мал. 1.

З побудованого прикладу видно, що приведений автомат не повністю визначений.

Означення. Скінчений автомат M називається детермінованим, якщо $\delta(q_i, a_k)$ містить не більше одного стану для любого $q_i \in Q$ та $a_k \in \Sigma$.

Твердження: Для довільного недетермінованого скінченого автомата M можна побудувати еквівалентний йому детермінований скінчений автомат M_1, такий що

$$L(M) = L(M_1).$$

Доведення: Нехай M – недетермінований скінчений автомат, такий що

$$M=< Q, \Sigma, \delta, q_0, F>$$

Детермінований автомат $M_1=< Q_1, \Sigma, \delta_1, q_{01}, F_1>$ побудуємо таким чином:

1. $Q_1 = P(Q)$, тобто імена станів автомата M1 – це підмножини множини Q.
2. $q_{01}= \{ q_0 \}$, $\{ q_0 \} \in P(Q)$.
3. F_1 складається з усіх таких підмножин $S \in P(Q)$, таких що $S \cap A \neq \varnothing$.
4. $\delta_1(S, a) = \{q \mid q \in \delta(q_i, a), q_i \in S \}$.

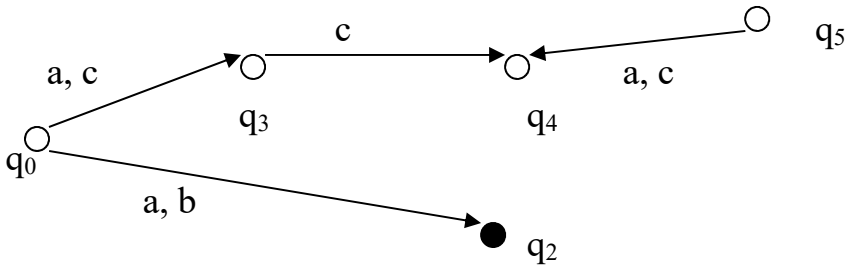
Доведемо індукцією по i, що $(S,w) \mid=^i (S_1,\varepsilon)$, тоді і тільки тоді, коли $S_1= \{ q \mid (q_i, w)) \mid=^i (q, \varepsilon)$, для $q_i \in S \}$,

Зокрема, $(\{q_0\}, w) \mid=^* (S_1, \varepsilon)$, для деякого $S_1 \in F_1$, тоді і тільки тоді, коли $(q_0, w) \mid=^* (q, \varepsilon)$, $q \in F$. Таким чином, $L(M) = L(M_1)$.

Побудований нами автомат M має дві властивості: він детермінований та повністю визначений. До того ж кількість станів цього автомата $2^n – 1$.

2.2 Мінімізація детермінованих скінчених автоматів

В подальшому при програмуванні скінчених автоматів важливо мати справу з так званими "мінімальними автоматами". Мінімальним для даного скінченого автомата візьмемо еквівалентний йому автомат з мінімальною кількістю станів. Те, що скінчені автомати можна мінімізувати покажемо на наступному прикладі:



Мал. 2.

Навіть при поверхневому аналізі діаграми переходів наведеного скінченого автомата видно, що вершини q_3, q_4 та q_5 є "зайвими", тобто при їх вилученні новий автомат буде еквівалентний початковому. З наведеного вище прикладу видно, що для отриманого детермінованого скінченого автомата можна запропонувати еквівалентний йому автомат з меншою кількістю станів, тобто мінімізувати скінчений автомат. Очевидно що серед зайвих станів цього автомата є недосяжні та тупикові стани.

Означення. Стан q скінченого автомата M називається недосяжним, якщо на діаграмі переходів скінченого автомата не існує шляху з q_0 в q.

Алгоритм. Пошук недосяжних станів. Спочатку спробуємо побудувати множину досяжних станів. Якщо Q_m - множина досяжних станів скінченого автомата M, то $Q \backslash Q_m$ - множина недосяжних станів. Побудуємо послідовність множин Q_0, Q_1, Q_2, … таким чином, що:

П0. $Q_0 = \{ q_0 \}$.

П1. $Q_1 = S_0 \cup \{ q \mid q \in \delta(q_0, a)$, для всіх $a \in \Sigma \}$.

П2. $Q_i = S_{i-1} \cup \{ q \mid q \in \delta(q_j, a), q_j \in Q_{i-1}$ та для всіх $a \in \Sigma \}$.

…….

Пm. $Q_m = Q_{m+1} =$ ……..

Очевидно, що кількість кроків П0, П1… скінчена, тому що послідовність Qi монотонна ($Q_0 \subseteq Q_1 \subseteq Q_2 \subseteq \ldots.$) та обмежена зверху - $|Q_m| <= |Q|$.

Тоді Q_m – множина досяжних станів скінченого автомата, а $Q \backslash Q_m$ – множина недосяжних станів.

Вилучимо з діаграми переходів скінченого автомата M недосяжні вершини. В новому (мінімізованому) автоматі функція δ визначається лише для досяжних станів. Побудований нами скінчений автомат з меншою кількістю станів буде еквівалентний початковому.

Означення. Стан q скінченого автомата M називається тупиковим, якщо на діаграмі переходів скінченого автомата не існує шляху з q в F.

Алгоритм. Пошук тупикових станів. Спочатку спробуємо знайти нетупикові стани. Якщо S_m - множина нетупикових станів, то $Q \backslash S_m$ - множина тупикових станів. Побудуємо послідовність множин S_0, S_1, S_2, … таким чином, що:

П0. $S_0 = \{ q \mid q \in F \}$.

П1. $S_1 = S_0 \cup \{ q \mid \delta(q, a) \cap S_0 \neq \varnothing, a \in \Sigma \}$.

П2. $S_i = S_{i-1} \cup \{ q \mid \delta(q, a) \cap S_{i-1} \neq \varnothing, a \in \Sigma \}$.

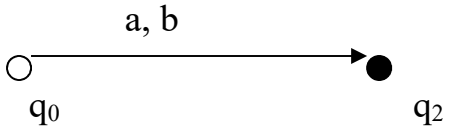
…….

Пм. $S_m = S_{m+1} = ……..$

Очевидно, що кількість кроків П0, П1… скінчена, тому що послідовність S_i монотонна ($S_0 \subseteq S_1 \subseteq S_2 \subseteq ….$) та обмежена зверху - $|S_m| <= |Q|$.

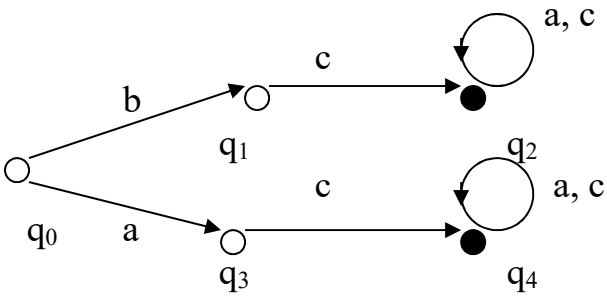
Тоді S_m – множина нетупикових станів скінченого автомата, а $Q \backslash S_m$ – множина тупикових станів. В новому (мінімізованому) автоматі функція δ визначається лише для нетупикових станів.

В прикладі наведеному на Мал. 2 множина недосяжних станів - $\{q_5\}$, а множина тупикових станів - $\{q_3, q_4\}$. Таким чином, для вище наведеного скінченого автомата еквівалентний йому автомат з меншою кількістю станів буде:



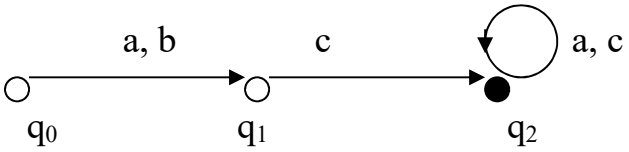
Мал. 3.

Автомат, у котрого відсутні недосяжні та тупикові стани, піддається подальшій мінімізації шляхом “склеювання” еквівалентних станів. Продемонструємо це на конкретному прикладі:



Мал. 4.

Очевидно, що для наведеного вище скінченого автомата можна побудувати еквівалентний йому скінчений автомат з меншою кількістю станів:



Мал. 5.

Ми досягли бажаного нам результату шляхом “склеювання” двох станів q_1, q_3. та q_2, q_4.

Означення. Два стани q_1 та q_2 скінченого автомата M називаються еквівалентними, якщо множини слів, які розпізнає автомат, починаючи з q_1 та q_2, співпадають.

Нехай q_1 та q_2 два різних стани скінченого автомата M, а $x \in \Sigma^*$. Будемо говорити, що ланцюжок x розрізняє стани q_1 та q_2, якщо $(q_1,x) |=^* (q_3, \varepsilon)$ та $(q_2,x) |=^* (q_4, \varepsilon)$, причому один з станів q_3 або q_4 не належить множині заключних станів. Стани q_1 та q_2 називаються k – нерозрізнені, якщо не існує ланцюжка x ($|x| <=k$), що розрізняє стани q_1 та q_2. Два стани q_1 та q_2 нерозрізнені, якщо вони к-нерозрізнені для довільного к.

Твердження: два стани q_1 та q_2 довільного скінченого автомата M з n станами нерозрізнені, якщо вони (n-2)-нерозрізнені.

Доведення: На першому кроці розіб'ємо множину станів скінченого автомата на дві підмножини: F та Q\F. На цій основі побудуємо відношення $\equiv^0$:
$q_1 \equiv^0 q_2$, якщо обидва стани одночасно належать F або Q\F.

Побудуємо відношення $\equiv^к$:
$q_1 \equiv^к q_2$, якщо $q_1 \equiv^{к-1} q_2$ та $\delta(q_1, a) \equiv^{к-1} \delta(q_2, a)$ для всіх $a \in \Sigma$.

Очевидно, кожна побудована множина містить не більше (n-1) елементи. Таким чином, можна отримати не більше (n-2) уточнення відношення $\equiv^0$.
Відношення $\equiv^{n-2}$ визначає класи еквівалентних станів автомата M.
Алгоритм. Побудова мінімального скінченого автомата.
П1. Побудувати скінчений автомат без тупикових станів.
П2. Побудувати скінчений автомат без недосяжних станів.
П3. Знайти множини еквівалентних станів та побудувати найменший (мінімальний) автомат.

Ознайомившись з деякими результатами теорії скінчених автоматів, спробуємо уяснити, які мови (словарні множини) є скінченоавтоматними.
Твердження: Скінчено автоматними є наступні множини:

- порожня словарна множина - $\varnothing$;
- словарна множина, що складається з одного ε-слова – {ε};
- множина {a}, $a \in \Sigma$.

Доведення: в кожному випадку нам доведеться конструктивно побудувати відповідний скінчений автомат:

1. Довільний скінчений автомат з пустою множиною заключних станів (а мінімальний – з пустою множиною станів) допускає $\varnothing$;
2. Розглянемо автомат $M = <\{q_0\}, \Sigma, q0, \delta, \{q0\}>$, у якому δ не визначено ні для яких $a \in \Sigma$.
   Тоді $L(M) = \{\varepsilon\}$.
3. Розглянемо автомат $M = <\{q_0, q_1\}, \Sigma, q_0, \delta, \{q_1\}>$, у якому функція δ визначена лише для пари (q_0, a), а саме: $\delta(q_0, a) = q_1$.
   Тоді $L(M) = \{a\}$.

Твердження: Якщо $M_1 = <Q_1, \Sigma, q_{01}, \delta_1, F_1>$ та $M_2 = <Q_2, \Sigma, q_{02}, \delta_2, F_2>$, що визначають відповідно мови $L(M_1)$ та $L(M_2)$, то скінченоавтоматними мовами будуть:

- $L(M_1) \cup L(M_2)$;
- $L(M_1) * L(M_2)$;
- $\{L(M_1)\} = \{\varepsilon\} \cup L(M_1) \cup L(M_1)* L(M_1) \cup \ldots.$,

де:

$$L(M_1) \cup L(M_2) = \{ w \mid w \in L(M_1) \text{ або } w \in L(M_2) \},$$
$$L(M_1) * L(M_2) = \{w=xy \mid x \in L(M_1),\ y \in L(M_2) \}.$$

Доведення:

1. Побудуємо автомат $M = <Q, \Sigma, q_0, \delta, F>$, такий що $L(M) = L(M_1) \cup L(M_2)$.
   - $Q = Q_1 \cup Q_2 \cup \{q_0\}$, де q_0 – новий стан ($q_0 \notin Q_1 \cup Q_2$);
   - Функцію δ визначимо таким чином:
     $\delta(q,a) = \delta_1(q,a)$, $q \in Q_1$, $a \in \Sigma$;
     $\delta(q,a) = \delta_2(q,a)$, $q \in Q_2$, $a \in \Sigma$;

$\delta(q_0,a) = \delta1(q_{01},a) \cup \delta2(q_{02},a)$, $q_{01} \in Q_1$, $q_{02} \in Q_2$, $a \in \Sigma$;

- Множина заключних станів $F = \begin{cases} F_1 \cup F_2, \text{якщо } \varepsilon \notin (L \cup L_2) \\ F_1 \cup F_2 \cup \{q_0\}, \text{яякщ } \varepsilon \in (L_1 \cup L_2) \end{cases}$

Побудований в цьому випадку автомат взагалі недетермінований. Індукцією по i >=1 покажемо, що $(q_0,w) |=^i (q,\varepsilon)$ можливо тоді і тільки тоді, коли $(q_{01},w) |=^i (q,\varepsilon)$, $q \in F_1$ або $(q_{02},w) |=^i (q,\varepsilon)$, $q \in F_2$.

2. Побудуємо автомат M = <Q, Σ, q_0, δ, F>, такий що L(M) = L(M_1) * L(M_2):
   - $Q = Q_1 \cup Q_2$;
   - $q_0 = q_{01}$;
   - Функцію δ визначимо таким чином:
     $\delta(q,a) = \delta_1(q,a)$, $q \in Q_1 \backslash F_1$, $a \in \Sigma$;
     $\delta(q,a) = \delta_2(q,a)$, $q \in Q_2$, $a \in \Sigma$;
     $\delta(q,a) = \delta_1(q,a) \cup \delta_2(q_{02},a)$, $q \in F_1$, $q_{02} \in Q_2$, $a \in \Sigma$;
   - Множина заключних станів $F = \begin{cases} F_2, \text{якщо } \varepsilon \notin L_2 \\ F_1 \cup F_2, \text{яякщ } \varepsilon \in L_2 \end{cases}$

3. Побудуємо автомат M = <Q, Σ, q_0, δ, F>, такий що L(M) = {L(M_1)}:
   - $Q = Q_1 \cup \{q_0\}$, де q_0 – новий стан ($q_0 \notin Q1$);
   - Функцію δ визначимо таким чином:
     $\delta(q,a) = \delta_1(q,a)$, $q \in Q_1 \backslash F_1$, $a \in \Sigma$;
     $\delta(q_0,a) = \delta_1(q_{01},a)$, $q_{01} \in Q_1$, $a \in \Sigma$;
     $\delta(q,a) = \delta_1(q,a) \cup \delta_1(q_{01},a)$, $q \in F_1$, $a \in \Sigma$;
   - Множина заключних станів $F = F_1 \cup \{q_0\}$.

2.3 Скінчені автомати та праволінійні граматики

Означення: Породжуюча граматика G – це п'ятірка

$$G=<N, \Sigma, P, S>,$$

де: N – скінчена множина - допоміжний алфавіт (нетермінали);
Σ – скінчена множина – основний алфавіт (термінали);
P – скінчена множина правил виду

$$\alpha \to \beta, \alpha \in (N \cup \Sigma)^* * N * (N \cup \Sigma)^*, \quad \beta \in (N \cup \Sigma).$$

S – виділений нетермінал (аксіома).

В залежності від структури правил граматики діляться на чотири типи (класифікація граматик по Хомському):

- Тип 0: граматики загального виду, коли правила не мають обмежень, тобто
  $$\alpha \to \beta, \alpha \in (N \cup \Sigma)^* * N * (N \cup \Sigma)^*, \quad \beta \in (N \cup \Sigma).$$
- Тип 1: граматики, що не укорочуються, коли обмеження на правила мінімальні, а саме:
  $$\alpha \to \beta, \alpha \in (N \cup \Sigma)^* * N * (N \cup \Sigma)^*, \quad \beta \in (N \cup \Sigma)^*, |\alpha| <= |\beta|.$$
- Тип 2: контекстно-вільні граматики, коли правила в схемі P мають вигляд:
  $$A_i \to \beta, \quad A_i \in N, \beta \in (N \cup \Sigma)^*.$$
- Тип 3: скінченоавтоматні граматики, коли правила в схемі P мають вигляд:

$A_i \to wA_j,\ A_i, A_j \in N, w \in \Sigma^*;$
$A_i \to w, w \in \Sigma^*;$
$A_i \to wA_j,\ A_i, A_j \in N, w \in \Sigma^*.$

В класі скінченоавтоматних граматик виділимо так звані праволінійні граматики – це граматики, які в схемі P мають правила виду :

$A_i \to wA_j,\ A_i, A_j \in N, w \in \Sigma^*;$
$A_i \to w, w \in \Sigma^*;$

Нескладно довести, що клас праволінійних граматик співпадає з класом граматик типу 3.

Означення: 1. Ланцюжок w_1 безпосередньо виводиться з ланцюжка w (позначається $w \Rightarrow w_1$), якщо $w = x\alpha y,\ w_1 = x\beta y$ та в схемі P граматики G є правило виду $\alpha \to \beta$. Оскільки поняття "безпосередньо виводиться" розглядається на парах ланцюжків, то в подальшому символ $\Rightarrow$ в подальшому буде трактуватися як бінарне відношення.

2. Ланцюжок w_1 виводиться з ланцюжка w (позначається $w \Rightarrow^* w_1$), якщо існує скінчена послідовність виду $w \Rightarrow w^1, W^1 \Rightarrow w^2, \ldots W^{n-1} \Rightarrow w_1$. Або кажуть, що бінарне відношення $\Rightarrow^*$ - це рефлексивно-транзитивне замикання бінарного відношення $\Rightarrow$.

3. Мова, яку породжує граматика G (позначається L(G)) – це множина термінальних ланцюжків:

$$L(G) = \{ w \mid S \Rightarrow^* w,\ w \in \Sigma^*\}.$$

Твердження: Клас мов, що породжуються праволінійними граматиками, співпадає з класом мов, які розпізнаються скінченими автоматами.

Доведення. Спочатку покажемо, що для довільної праволінійної граматики G можна побудувати скінчений автомат M, такий що L(M) = L(G).

Розглянемо правила праволінійної граматики. Вони бувають двох типів:

- $A_i \to wA_j,\ A_i, A_j \in N, w \in \Sigma^*;$ (1)
- $A_i \to w, w \in \Sigma^*;$ (2)

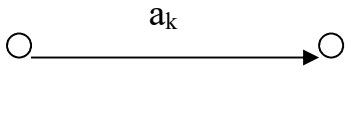
На основі правил граматики G побудуємо схему P1 нової граматики, яка буде еквівалентною початковій, а саме:

- правила виду $A_i \to a_1a_2\ldots a_pA_j$ замінимо послідовністю правил:
  $A_i \to a_1B_1$
  $B_1 \to a_2B_2$
  .....
  $B_{p-1} \to a_pA_j$;
- правила виду $A_i \to a_1a_2\ldots a_p$ замінимо послідовністю правил:
  $A_i \to a_1B_1$
  $B_1 \to a_2B_2$
  .....
  $B_{p-1} \to a_p B_p$
  $B_p \to \varepsilon$

де: $B_1,\ B_2,\ \ldots$ - це нові нетермінали граматики G_1. Очевидно, що граматика G_1 буде еквівалентна граматиці G.

Далі, на основі граматики G1 побудуємо скінчений автомат M, таким чином:

- як імена станів автомата візьмемо нетермінали граматики G1;
- початковий стан автомата позначається аксіомою S;
- функція δ визначається діаграмою переходів, яка будується на основі правил виду $A_i \rightarrow a_k A_j$:



- множина F заключних станів скінченого автомата визначається так:

$$F = \{A_i \mid A_i \rightarrow \varepsilon\}.$$

Індукцією по довжині вхідного слова покажемо, що якщо $S \Rightarrow^{n+1} w$, то $(q_0, w) \mid=^n (q, \varepsilon)$:

Базис $i = 0$: $S \Rightarrow \varepsilon$, тоді $(q_0, \varepsilon) \mid=^0 (q_0, \varepsilon)$.

Нехай $|w| = i+1$, тобто $w = aw_1$: тоді $S \Rightarrow aA_p \Rightarrow^i aw_1$ та

$$(q_0, aw_1) \mid= (q_i, w_1) \mid=^{i-1} (q, \varepsilon),\ q \in F.$$

Доведення навпаки є очевидним.

2.4 Регулярні множини та регулярні вирази

Означення: Нехай Σ - скінчений алфавіт. Регулярна множина в алфавіті Σ визначається рекурсивно:

П1. ∅ - пуста множина – це регулярна множина в алфавіті Σ;

П2. {ε} – пусте слово – регулярна множина в алфавіті Σ;

П3. {a} – однолітерна множина – регулярна множина в алфавіті Σ;

П4. Якщо P та Q – регулярні множини, то такими є наступні множини:

P ∪ Q (операція об'єднання);

P * Q (операція конкатенації);

{P} = {ε} ∪ P ∪ P*P ∪ ……. (операція ітерації).

П5. Ніякі інші множини, окрім побудованих на основі ПП 1-4 не є регулярними множинами.

Таким чином, регулярні множини можна побудувати з базових елементів ПП 1-3 шляхом скінченого застосування операцій об'єднання, конкатенації та ітерації.

Регулярні вирази позначають регулярні множини таким чином, що:

П1. 0 позначає регулярну множину ∅;

П2. ε позначає регулярну множину {ε};

П3. a позначає регулярну множину {a};

П4. Якщо p та q позначають відповідно регулярні множини P та Q, то

p + q позначає регулярну множину P ∪ Q;

p · p позначає регулярну множину P * Q;

p* позначає регулярну множину {P}.

П5. Ніякі інші вирази, окрім побудованих на основі ПП 1-4 не є регулярними виразами.

Оскільки ми почали вести мову про вирази, нам зручніше перейти до поняття алгебри регулярних виразів. Для кожної алгебри одним з важливих питань є

питання тотожних перетворень, які виконуються на основі тотожностей в цій алгебрі. Сформулюємо основні тотожності алгебри регулярних виразів:

1) $a + b = b + a;$
2) $a\varepsilon = \varepsilon a = a;$
3) $a0 = 0a = 0;$
4) $a(b + c) = ab + ac;$
5) $(a + b)c = ac + bc;$
6) $a + a = a;$
7) a + b + c = a + (b + c);
8) $a + b + c = (a + b) + c;$
9) $p + p^* = p^*;$
10) $0^* = \varepsilon;$
11) $\varepsilon^* = \varepsilon.$

По аналогії з класичними алгебрами розглянемо лінійне рівняння в алгебрі регулярних виразів:

$X = aX + b,$ де a, b – регулярні вирази.

Взагалі таке рівняння в залежності від a та b може мати безліч розв'язків, наприклад: $X = a^*(b + \gamma),$ при умові, що а позначає ε-ланцюжок. Тоді γ може бути навіть і нерегулярним виразом. Серед всіляких розв'язків рівняння з регулярними коефіцієнтами виберемо найменший розв'язок $X = a^*b$, який назвемо <u>*найменша нерухома точка*</u>. Щоб перевірити, що a^*b розв'язок рівняння в алгебрі регулярних виразів, підставимо його в початкове рівняння та перевіримо тотожність виразів на основі системи тотожних перетворень:

$a^*b = aa^*b+b = (aa^*+\varepsilon)b = (a(\varepsilon + a + aa+...) + \varepsilon)b = (\varepsilon + a + aa +...)b = a^*b$

В алгебрі регулярних виразів також розглядають і системи лінійних рівнянь з регулярними коефіцієнтами:

$X_1 = a_{11}X_1 + a_{12}X_2 + ... + a_{1n}X_n + b_1$
$X_2 = a_{21}X_1 + a_{22}X_2 + ... + a_{2n}X_n + b_2$
……
$X_n = a_{1n}X_1 + a_{2n}X_2 + ... + a_{nn}X_n + b_n$
де $a_{ij}, b_i, i,j = 1..n.$

Метод розв'язку системи лінійних рівнянь з регулярними коефіцієнтами нагадує метод виключення Гауса.

П1. Покласти $i = 1.$

П2. Якщо i рівне n, то перейти на П4. Інакше i–*е* рівняння, використавши систему тотожних перетворень, записати у вигляді:

$X_i = aX_i + b,$ де a - регулярний вираз в алфавіті Σ, а b – регулярний вираз виду $\beta_0 + \beta_{i+1}X_{i+1} + \beta_{i+2}X_{i+2} + ... \beta_nX_n,$ де β_k (k=0, i+1, .. n) - регулярні коефіцієнти. Далі, в правих частинах рівнянь зі змінними $X_{i+1}, X_{i+2}, ... X_n$ в лівій частині рівняння підставити заміст X_i значення a^*b.

П3. Збільшити і на 1 та перейти на П2.

П4. Записати рівняння $X_n = aX_n + b$, де a, b – регулярні вирази в алфавіті Σ. Це можливо оскільки на кожному кроці виконання П2 для кожного i < n в в правій частині рівняння для X_i не буде невідомих $X_1, X_2, ..., X_{i-1}$. Перейти на П5, при цьому і буде рівне n.

П5. Для рівняння з X_i, котре має вигляд $X_i = aX_i + b,$ де a, b - регулярні вирази в алфавіті Σ, записати $X_i = a^*b$. В рівняння для $X_1, X_2, ..., X_{i-1}$ підставити замість X_i регулярний вираз a^*b.

П6. Якщо і рівне 1 то зупинитися, інакше і зменшити на 1 та перейти на П5.

Приклад 4. Розв'язати систему лінійних рівнянь:

$$X = a_1X + a_2Y + a_3$$
$$Y = b_1X + b_2Y + b_3$$

З першого рівняння:

$$X = a_1*(a_2Y + a_3).$$

Підставимо значення X в друге рівняння та зведемо подібні члени:

$$Y = b_1a_1*(a_2Y+a_3) + b_2Y + b_3 = b_1a_1*a_2Y + b_1a_1*a_3 + b_2Y + b_3 =$$
$$(b_1a_1*a_2 + b_2)Y + (b_1a_1*a_3 + b_3)$$

Таким чином,

$$Y = (b_1a_1*a_2 + b_2)* (b_1a_1*a_3 + b_3).$$

Підставимо Y в перше рівняння:

$$X = a_1X + a_2(b_1a_1*a_2 + b_2)* (b_1a_1*a_3 + b_3) + a_3$$

Звідси,

$$X = a_1*(a_2(b_1a_1*a_2 + b_2)* (b_1a_1*a_3 + b_3) + a_3)$$

2.5 Польський інверсний запис для регулярних виразів

Польський інверсний запис (ПОЛІЗ) для регулярних виразів будується на основі початкового регулярного виразу на основі наступних правил:

1. Порядок операндів в початковому виразі і в перетвореному виразі співпадають.
2. Операції в перетвореному виразу йдуть з урахуванням пріоритету безпосередньо за операндами.

Наприклад, ПОЛІЗ для виразу *(a*+b)*c* має такий вигляд: *a*b+*c*· .

В цьому прикладі в стандартному записі регулярного виразу бінарна операція конкатенація (·) природньо опущена, але в ПОЛІЗ потрібно завжди цю операцію явно вказувати. Важливою характеристикою ПОЛІЗ є відсутність дужок в запису виразу.

Алгоритм. Перетворення регулярного виразу в ПОЛІЗ.

Для перетворення виразу в ПОЛІЗ необхідно з кожною операцією зв'язати деяке число, яке будемо називати "пріоритет" (0 – найвищий пріоритет присвоїмо дужці '(').

П0. Прочитати поточну лексему.

П1. Якщо поточна лексема операнд, то занести її в поле результату та перейти на П0.

П2. Якщо поточна лексема '(', то занести її в стек та перейти на П0.

П3. Якщо поточна лексема код операції, то доки пріоритет операції на вершині стека більший або рівний пріоритетові поточної операції, елементи з вершини стека перенести в поле результату. Поточну лексему занести в стек та перейти на П0.

П4. Якщо поточна лексема ')', то з вершини стека в поле результату перенести всі коди операцій, які передують в стеку ')'. Дужку '(' зняти з вершини стека та перейти на П0.

П5. Якщо досягли кінця вхідного виразу, то всі елементи із стека перенести в поле результату.

2.6 Інтерпретація ПОЛІЗ регулярного виразу.

Результат інтерпретації ПОЗІЗ – це скінчений автомат М, який розпізнає (сприймає) множину ланцюжків, котрі позначає регулярний вираз.

П0. Причитати поточну лексему.

П1. Якщо поточна лексема операнд a_i, то побудувати скінчений автомат М, такий що $L(M) = \{a_i\}$. Занести автомат в стек (якщо не повністю автомат, то по меншій мірі в стек занести вказівник на цей автомат). Перейти на П0.

П2. Якщо поточна лексема код операції, то по її арності з вершини стека зняти автомати, побудувати результуючий автомат М в залежності від операції:

а) для операції '+':

$L(M)=L(M_1) \cap L(M_2)$, де $L(M_1)$ та $L(M_2)$ - скінчені автомати, які знаходились на вершині стека;

б) для операції '.':

$L(M)=L(M_2) * L(M_1)$, де $L(M_1)$ та $L(M_2)$ - скінчені автомати, які знаходились на вершині стека;

в) для операції '*':

$L(M)= \{L(M_1)\}$, де $L(M_1)$ - скінчений автомати, що знаходився на вершині стека;

Побудований нами автомат занести в стек. Перейти на П0.

П3. Якщо досягли кінця регулярного виразу, то на вершині стека знаходиться автомат *M*, який розпізнає множину слів (ланцюжків), які позначає регулярний вираз.

Завдання. Для регулярного виразу *ab+** побудувати скінчений автомат, який розпізнає множину ланцюжків, що позначаються цим виразом.

2.7 Застосування скінчених автоматів при розробці лексичних аналізаторів

Поділ множини лексем мови програмування на класи можна виконати на основі різних критеріїв, наприклад, множину операцій в мові С можна розбити на три класи:

- Однолітерні коди операцій: +, -, *, / …;
- Двохлітерні коди операцій: >=, ++, --, …;
- Трилітерні коди операцій: >>=, <<=, ….

Формальне визначення класу лексем мови програмування можна виконати одним з нижченаведених способів:

- За допомогою праволінійних граматик;
- За допомогою скінчених автоматів;
- За допомогою регулярних виразів;
- Перерахувати лексеми даного класу як скінчену множину елементів;

Перші три способи визначення класів лексем за своєю потужністю еквівалентні. Якщо деякий клас лексем мови програмування скінчена множина, то одним з тривіальних способів визначення лексем цього класу є їх перерахування. Наприклад, клас однолітерних кодів операцій мови програмування *C* можна визначити як скінчену множину

L_0 = { +, -, /, *, ….}

Сформулюємо фундаментальне твердження теорії граматик та автоматів: клас мов, які розпізнаються скінченими автоматами, співпадає з класом мов визначених праволінійними граматиками та регулярними виразами та навпаки. Відмітимо, що аналіз досвіду використання перерахованих засобів визначення класів лексем показує, що скінчені автомати знайшли широке використання при розробці лексичних аналізаторів для конкретних мов програмування, а регулярні вирази та праволінійні граматики широко використовуються в системах автоматизації побудови мовних процесорів як засоби високого рівня денотативності опису класів лексем.

Розглянемо можливі варіанти використання різних підходів визначення класів лексем мов програмування та їх реалізацію в мовних процесорах. Продемонструємо два підходи розробки програмних модулів, які розпізнають множину ланцюжків (лексем), що допускає скінчений автомат, діаграма переходів котрого зображена на Мал. 1.

Скінчений автомат має:

- множину станів Q={q_0, q_1, q_2, q_3, q_4, q_5, .., q_{17}};
- вхідний алфавіт Σ={0,1,..9, A, B,..F, a, b,..f, X, x, L, l, U, u};
- початковий стан q_0;
- множину заключних станів F= Q\{ q_0, q_{12}};
- відображення δ визначено діаграмою переходів.

Програму, яка розпізнає множину лексем, реалізуємо двома способами:

- <u>перший спосіб</u>: створимо програмний модуль, роботою котрого керує таблиця управління скінченого автомата $M(q_i, a_j)$, яка визначається в програмі явно;
- <u>другий спосіб</u>: розробимо програмний модуль з управлінням за номером поточного стану скінченого автомата та поточною вхідною літерою.

Програмний модуль з управлінням на основі таблиці $M(q_i, a_j)$:

Реалізуємо програмний модуль для скінченого автомата, діаграма переходів котрого зображена на Мал. 1.

```
#include <stdio.h>
#define OK  1
#define ERROR    0
char alphabet[ ]="0123456789ABCDEFabcdefXxLlUu"; // кількість елементів - 28
//          0, 1, 2, 3, 4, 5, 6, 7,  8, 9, A, B, C, D, E, F, a, b, c, d, e, f, X, x, L, l U, u
int sigma[ ][ ]= {
 {6, 2, 2, 2, 2, 2, 2, 2, 2, 2,  0, 0,  0, 0,  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },  // q0=1
 {2, 2, 2, 2, 2, 2, 2, 2, 2, 2,  0, 0,  0, 0,  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 3, 5, 5 },  // q1=2
  {0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  0, 0,  0, 0,  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 4 },  // q2=3
 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  0, 0,  0, 0,  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0  },  // q3=4
 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  0, 0,  0, 0,  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 6, 0, 0  },  // q4=5
 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  0, 0,  0, 0,  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0  },  // q5=6
 {8, 8, 8, 8, 8, 8, 8, 8, 0, 0,  0, 0,  0, 0,  0, 0, 0, 0, 0, 0, 0, 0, 0, 13, 13, 0, 0, 0, 0  },  // q6=7
```

```
 {8, 8, 8, 8, 8, 8, 8, 8, 0, 0,  0, 0,  0, 0,  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 9, 9, 11, 11  },  //q7=8
 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  0, 0,  0, 0,  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 10, 10  },  //q8=9
 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  0, 0,  0, 0,  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0  },  //q9=10
 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  0, 0,  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 12, 12, 0, 0  },
 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  0, 0,  0, 0,  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0  },  //q11=12
 {14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,0,0,0,0,0,0}
//q12=13
{14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,0,0,17,17,15,15},
 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  0, 0,  0, 0,  0, 0, 0, 0, 0, 0, 0, 0, 0, 0,16, 16, 0, 0},  //q14=15
 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  0, 0,  0, 0,  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0  },  //q15=16
 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  0, 0,  0, 0,  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 18, 18  },  //q16=17
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  0, 0,  0, 0,  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0  }  //q17=18
        };
//Якщо σ(qi,aj)-невизначено, то M(qi,aj)=0
int finish[ ]={1,2,4,5,7,8,9,10,11,12,13,14,15,16,17,0}; //заключні стани автомата
//функція, яка по вхідній літері вказує її індекс в масиві alphabet
int  index_litera (int litera)
       { for(int i=0; *(alphabet+i)  ;i++)
                if (litera = = *(alphabet+i)) return i;
       return -1;  // помилка: недопустима літера
       }
is_final (int row )
   {  for (int i = 0; *( finish+i) ; i++)  if (*( finish+i) == row ) return 1;
         return 0;
         )
char text[80];
int  * MASSIVE(sigma, row, column)
   { if  (row == 0) return NULL  else return  (sigma+(row-1)*28+colomn); }
int automat (FILE*fp)
{int row, c, column, id=0; //row-ім'я рядка таблиці, column - ім'я стовпчика
  *(text+id++)=0;   row=0; // початкові установки
    while((c=fgetch(fp))!=EOF) {
         if ((column = index_litera(c)) = = ERROR) break;
           row=*MASSIV(sigma, row, column);
          *(text+id++)=c; *(text+id)=0;
           if (row = = ERROR) break;
         }
//досягли кінця файла, або недопустима літера, або…….
if (is_final ( row )) //ми знаходимося в заключному стані
         return OK; else retutn ERROR;
}

main(int argc, char* argv[ ])
{FILE*fp; char file_name[80];
REPEAT:
```

```
printf("Вкажіть ім'я файла з лексеми:");
if(scant("%s", file_name) == 0) return 0; //відмовляється працювати
if( fp = fopen (file_name,"rt") == NULL)
        { printf ("файл %s не відкрито.\ n"); goto REPEAT;}
while (! eof (fp))
         if ( automat(fp) == ERROR) printf (" Лексема вірна - %s", text);
                 else  printf ("Лексема помилкова-%s", text);
}// кінець main
```

Нехай маємо текстовий файл, в якому знаходяться лексеми, які відділяються одна від іншої такими символами як: проміжок, \n, \v, \r, \t.. щоб зменшити об'єм таблиці sigma, пропуск символів, які розділяють лексеми в текст будемо виконувати додатково, цілком у функції automat.

Очевидною перевагою програмної реалізації скінчених автоматів наведеним вище способом є простота визначення інформаційних масивів даних: alphabet — вхідний алфавіт автомата, sigma — таблиця переходів автомата, finish — масив імен заключних станів. Серед недоліків реалізації наведеної вище програми яскраво виділяється один – досить великі затрати пам'яті для таблиці sigma. Очевидно, що матриця sigma сильно розріджена, тобто більшість її елементів має значення ERROR. В такому випадку при реалізації скінчених автоматів, кількість станів яких вимірюється десятками, та як вхідний алфавіт розглядається вся кодова таблиця ЕОМ (255 символів), то затрати оперативної пам'яті будуть занадто великі.

Програмна реалізація скінченого автомату з управлінням на основі поточного стану.

Скористаємося можливостями конструкції switch мови програмування С. Як елемент управління скінченого автомата візьмемо поточний стан скінченого автомата (на початку програми q0=0).

```
int automat (FILE*fp)
{ int c, q=0; // початковий стан автомата
  while((c=fgetch(fp))!=EOF)
    { if (index_litera(c) = = -1) break;
      switch(q)
      {case 0:
              if (c>='1' && c<='9") {q=1; break;}
              if (c= =' ' || c= ='\n' || c= ='\t' || c= ='\r') break;
              if (c= ='0') { q=6; break;}
       // помилка – недопустимий початок лексеми
              my_error( ); return 0;
       case 1:
              if (c>='0' && c<='9') break;
              // в заключному стані
              return 1;
      case 6:
              if (c= ='X' || c= ='x') {q=12; break;}
              if (c>='0' && c<='7') {q=7; break;}
```

```
            // в заключному стані
            return 1;
       case 12:
            if ((c>='0' && c<='9') ||
                (c>='A' && c<='F') ||
                (c>='a' && c<='f'))     {q=13; break;}
      // помилка – невірне продовження шістнадцяткової константи
                 my_error ( ); return 0;
        // ---------------------------------------------------------------
        // в такому стилі програмуємо далі
        // ---------------------------------------------------------------
    }// кінець конструкції switch
  // перевіримо на заключний стан
 if (is_final ( row )) printf (" Лексема вірна - %s", text);
                else  printf ("Лексема помилкова-%s", text);
  }// кінець циклу зчитування літер
}// кінець функції automat
```

Наведений вище приклад функції automat, в якій в основу управління покладено поточний стан скінченого автомата, спробуємо певною мірою оптимізувати. З тексту модуля видно, що основні операції – це операції порівняння та перевірка належності поточної вхідної літери діапазону літер. При цьому враховується конкретна властивість кодової таблиці ASCI: коди латинських літер та цифр займають певні діапазони. Щоб функція automat не залежала від властивостей кодової таблиці, запропонуємо новий варіант її реалізації.

Розділимо літери кодової таблиці на класи:

- проміжок, \n, \v, \r, \t, \b – літери, які розділяють лексеми;
- 1, 2, 3, ..7 – вісімкові цифри (крім 0);
- 0,...9,A,..F,a,..f — шістнадцяткові цифри;
- 0 – вісімковий або десятковий нуль
- клас, до якого входять літери ASCI: \, Σ .

Як бачимо, в загальному випадку класи можуть перетинатися.

Розглянемо таблицю, об'ємом 256 елементів, в якій буде відмічатися належність літер до окремих класів.

```
# include <stdio.h>
#define DECIMAL        0x01
#define OCTAL          0x02
#define EMPTY          0x80
#define  LITERA        0x04
char decimal[]='0123456789';
char hesh_digit[] = '0123456789ABCDEFabcdef';
char octal[] =  '01234567';
char empty[]  =  '  \n\r\t\v\r\b';
char litera[]=
        'ABCDEFGHIJKLMNOPQRSTUVUXYZ_abcdefghijklmnopqrstuvuxyz'
char vector_upr [256];
```

```
// початкова ініціалізація
void set_class (char * str, int code)
  { int i, len = strlen(str);
    for (i=0; i<len; i + +)  vector_upr [(int)*(str+i)]  |=  code;
  }
int automat (FILE * fp)
{ int c, q =0; // початковий стан автомата
  while ((c = fgetch(fp))! = EOF)
   { class_litera = vector_upr [(int) ( c )];
     if (!class_litera) break; // літера не належить множині sigma
     switch (q)
      { case 0: if (class_litera & DECIMAL) { q=1; break;}
                if (c == '0')  {q=6; break; }
                if (class_litera & EMPTY) break;
                return ERROR;
        // в такому стилі програмуємо далі
        … …. … …..
        }
    }  }
main ( ) {
        int  i;
        FILE*fp; char file_name[80];
        REPEAT:
        printf("Вкажіть ім'я файла з лексеми:");
        if(scant("%s", file_name) == 0) return 0; //відмовляється працювати
        if( fp = fopen (file_name,"rt") == NULL)
                { printf ("файл %s не відкрито.\ n"); goto REPEAT;}
        for (i = 0; i < 256;  i++)  *(vector_upr + i) = 0;
        set_class(decimal, DECIMAL);
        set_class(octal, OCTAL);  set class(litera, LITERA);
        set class(empty, EMPTY);
        while (! eof (fp))
                if ( automat(fp) == ERROR) printf (" Лексема вірна - %s", text);
                        else  printf ("Лексема помилкова-%s", text);
      }// кінець main
```

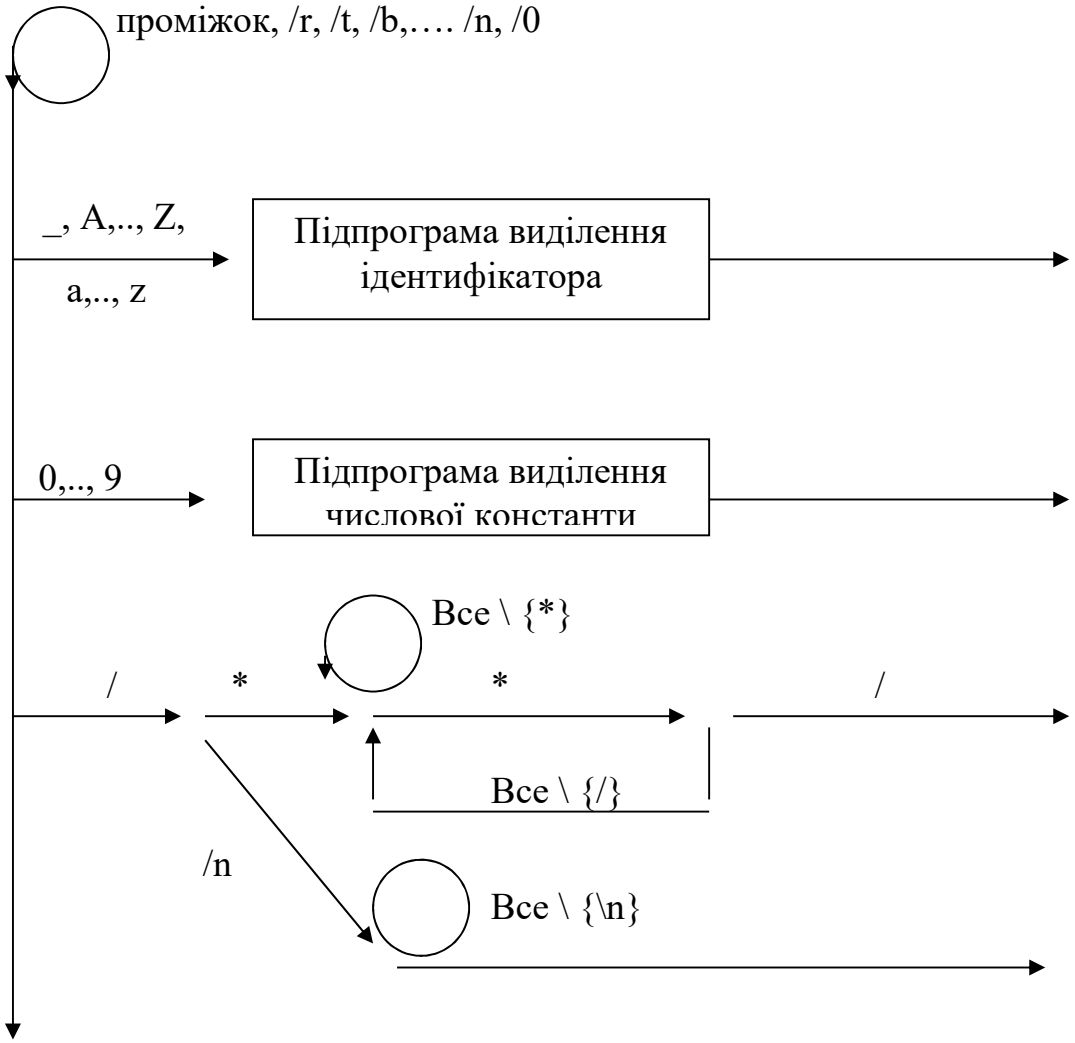
2.8 Методика програмування лексичних аналізаторів на основі скінчених автоматів.

Продемонструємо використання методики програмування лексичних аналізаторів на прикладі лексичного аналізатора для мови програмування С. Множину лексем мови програмування С розіб'ємо на наступні класи:

- зарезервовані слова (підмножина ідентифікаторів);
- цілочислові константи (тип int);
- дійсні константи (типи double, float);
- літерні константи (тип char);
- рядкові константи (масиви елементів базового типу char);

- коментарі;
- коди операцій та спеціальні символи;
- символи, які не мають семантичної інтерпретації, окрім як роздільники лексем (проміжок, \n, \v, \r, \t, \b, \0 тощо).

Побудуємо діаграму переходів об'єднаного скінченого автомата для лексичного аналізатора мови програмування С – Мал. 7.



.....
.....

Далі продовжуємо у тому ж стилі.

Мал. 7.

Після виділення нової лексеми програмний модуль переходить у початковий стан. Перед звертанням до функції, яка виділяє лексему з вхідного тексту, необхідно виконати функцію ungetch(fp).

Окрім виділення лексем з вхідного тексту програми лексичний аналізатор повинен виконувати функцію локалізації лексичних помилок. Функція локалізації лексичної помилки повинна "пропустити" фрагмент тексту вхідної програми мінімальної довжини так, щоб подальша робота була продовжена. Очевидно, що таку умову задовольняють літери, по яких лексичний аналізатор переходить з початкового стану в інші. З діаграм переходів видно, що це – літери з множини { проміжок, \n, \v, \r, \t, \b, +, -, …}.

3. Лабораторний практикум побудови лексичних аналізаторів.

1. Розробити та реалізувати представлення скінченого автомата в пам`яті ЕОМ. Реалізувати алгоритм перетворення недетермінованого скінченого автомата в еквівалентний йому детермінований скінчений автомат.

2. Розробити та реалізувати представлення скінченого автомата в пам`яті ЕОМ. Реалізувати алгоритм мінімізації детермінованого скінченого автомата.

3. Розробити та реалізувати представлення скінченого автомата в пам`яті ЕОМ. Реалізувати алгоритм пошуку слова мінімальної довжини, що допускається двома скінченими автоматами.

4. Розробити та реалізувати представлення скінченого автомата в пам`яті ЕОМ. Розробити та реалізувати алгоритм перевірки чи допускає скінчений автомат слова виду $\omega=\omega_0\omega_1$, де ω_0 - наперед задане (фіксоване) слово.

5. Розробити та реалізувати представлення скінченого автомата в пам`яті ЕОМ. Розробити та реалізувати алгоритм перевірки чи допускає скінчений автомат слова виду $\omega=\omega_1\omega_0$, де ω_0 - наперед задане (фіксоване) слово.

6. Розробити та реалізувати представлення скінченого автомата в пам`яті ЕОМ. Розробити та реалізувати алгоритм перевірки чи допускає скінчений автомат слова виду $\omega=\omega_1\omega_0\omega_2$, де ω_0 - наперед задане (фіксоване) слово.

7. Розробіть алгоритм та реалізуйте програму, що моделює роботу недетермінованого скінченого автомата.

8. Реалізуйте лексичний аналізатор мови програмування С. Для зберігання класів лексем організуйте таблиці. Виведіть вміст таблиць після обробки тексту програми.

9. Реалізуйте лексичний аналізатор мови програмування С++. Для зберігання класів лексем організуйте таблиці. Виведіть вміст таблиць після обробки тексту програми.

10. Реалізуйте лексичний аналізатор мови програмування Pascal. Для зберігання класів лексем організуйте таблиці. Виведіть вміст таблиць після обробки тексту програми.

11. Реалізуйте лексичний аналізатор мови програмування Turbo Pascal 7.xx. Для зберігання класів лексем організуйте таблиці. Виведіть вміст таблиць після обробки тексту програми.

12. Реалізуйте лексичний аналізатор мови програмування Delphi. Для зберігання класів лексем організуйте таблиці. Виведіть вміст таблиць після обробки тексту програми.

13. Реалізуйте лексичний аналізатор мови програмування FORTRAN. Для зберігання класів лексем організуйте таблиці. Виведіть вміст таблиць після обробки тексту програми.

4. Синтаксичний аналіз в мовних процесорах

Для визначення синтаксичної компоненти мови програмування використовують контекстно-вільні граматики (КС-граматики). На відміну від скінчено-автоматних граматик потужність класу КС-граматик достатня, щоб визначити майже всі так звані синтаксичні властивості мов програмування. Якщо цього недостатньо, то розглядають деякі спрощення у граматиках типу 2 або параметричні КС-граматики.

Звичайно, із синтаксичною компонентою мови програмування пов'язана семантична компонента. Тоді, якщо ми говоримо про семантику мови програмування, ми вимагаємо семантичної однозначності для кожної вірно написаної програми. За аналогією з семантикою, при описі синтаксичної компоненти мови програмування необхідно користуватися однозначними граматиками.

Означення. Граматика G називається неоднозначною, якщо існує декілька варіантів виводу ω в G ($\omega \in L(G)$).

Приклад. Розглянемо таку граматику $G = \langle N, \Sigma, P, S \rangle$ зі схемою *P*.

$S \to S + S \mid S * S \mid a$

Покажемо, що для ланцюжка $\omega = a + a + a$ існує щонайменше два варіанти виводу:

1. $S \Rightarrow S + S \Rightarrow S + S + S \Rightarrow a + S + S \Rightarrow a + a + S \Rightarrow a + a + a$
2. $S \Rightarrow S + S \Rightarrow a + S \Rightarrow a + S + S \Rightarrow a + a + S \Rightarrow a + a + a$

В теорії граматик розглядається декілька стратегій виведення ланцюжка ω в *G*. Визначимо дві стратегії, які будуть використані в подальшому.

Лівостороння стратегія виводу ланцюжка ω в *G* — це послідовність кроків безпосереднього виводу, при якій на кожному кроку до уваги береться перший зліва направо нетермінал. *Правостороння стратегія виводу* ω в *G* протилежна лівосторонній стратегії. З виводом ω в *G* пов'язане синтаксичне дерево, яке визначає синтаксичну структуру програми.

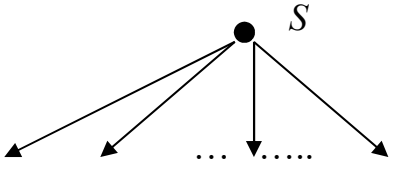
Означення. Синтаксичне дерево виводу ω в *G* — це впорядковане дерево, корінь котрого позначено аксіомою, в проміжних вершинах знаходяться нетермінали, а на кроні — елементи з $\Sigma \cup \{\varepsilon\}$. Побудова синтаксичного дерева виведення ω в G виконується покроково з урахуванням стратегії виводу ω в G.

Алгоритм. Побудова синтаксичного дерева ланцюжка ω в граматиці G з урахуванням лівосторонньої стратегії виводу:

$П_1$. Будуємо корінь дерева та позначимо його аксіомою *S*.

$П_2$. В схемі P граматики G візьмемо правило виду $S \to \alpha_1 \alpha_2 ... \alpha_p$, де $\alpha_i \in N \cup \Sigma \cup \{\varepsilon\}$. Та побудуємо дерево висоти 1 (Мал. 1).

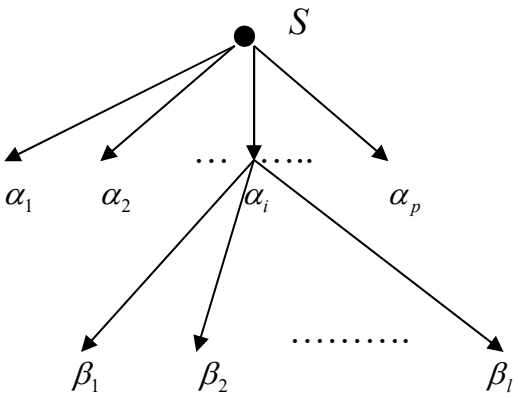
$П_i$. На кроні дерева, побудованого на попередньому кроку, візьмемо перший зліва направо нетермінал. Нехай це буде α_i. Тоді в схемі P виберемо правило виду $\alpha_i \to \beta_1 \beta_2 ... \beta_l$, де $\beta_i \in N \cup \Sigma \cup \{\varepsilon\}$ та побудуємо наступне дерево (Мал. 2).



α_1 α_2 α_i α_p Мал. 1.

Пункт $П_i$ виконувати доти, доки на кроні дерева будуть елементи з N.
Відмітимо очевидні факти, що випливають з побудови синтаксичного дерева:

- крона дерева, зображеного на мал. 2 наступна $\alpha_1\alpha_2...\alpha_{i-1}\beta_1\beta_2...\beta_l\alpha_{i+1}...\alpha_p$. Ланцюжок з крони $\alpha_1\alpha_2...\alpha_{i-1} \in \Sigma^*$ - це термінальний ланцюжок.
- для однозначної граматики G існує лише одне синтаксичне дерево виводу ω в G.



Мал. 2.

Означення. Будемо говорити, що ланцюжок $\omega \in \Sigma^*$, побудований на основі граматики $G(\omega \in L(G))$ *проаналізований*, якщо відоме одне з його дерев виводу. Зафіксуємо послідовність номерів правил, які були використані під час побудови синтаксичного дерева виводу ω в G з урахуванням стратегії виводу.

Означення. *Лівостороннім аналізом* π ланцюжка $\omega \in L(G)$ будемо називати послідовність номерів правил, які були використані при лівосторонньому виводі ω в *G*.

Приклад: Для граматики $G = \langle N, \Sigma, P, S \rangle$ зі схемою Р

$S \to S + T$ (1)

$S \to T$ (2)

$T \to T * F$ (3)

$T \to F$ (4)

$F \to (S)$ (5)

$F \to a$ (6)

для ланцюжка $\omega = a*(a+a)$ побудуємо лівосторонній аналіз π:

$S \Rightarrow T \Rightarrow T*F \Rightarrow F*F \Rightarrow a*F \Rightarrow a*(S) \Rightarrow a*(S+T) \Rightarrow a*(T+T) \Rightarrow$

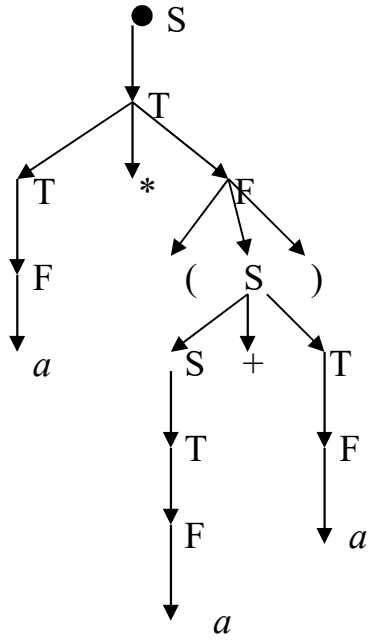
$a*(F+T) \Rightarrow a*(a+T) \Rightarrow a*(a+F) \Rightarrow a*(a+a)$

З наведеного вище виводу ланцюжка $\omega \in L(G)$ лівосторонній аналіз π буде: $\pi = \langle 2,3,4,6,5,1,2,4,6,4,6 \rangle$, а синтаксичне дерево виводу $\omega = a*(a+a)$ зображено на Мал. 4.

Нехай π лівосторонній аналіз ланцюжка $\omega \in L(G)$. Знаючи π досить легко побудувати (відтворити) синтаксичне дерево. Відтворення (синтез) синтаксичного дерева можна виконати, скориставшись однією з стратегій синтаксичного аналізу:

- стратегія "зверху донизу";

- стратегія "знизу вгору".



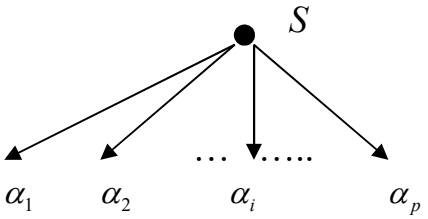
Мал. 4

Стратегія синтаксичного аналізу "зверху донизу" – це побудова синтаксичного дерева крок за кроком починаючи від кореня до крони.

Алгоритм 2. Синтез синтаксичного дерева на основі лівостороннього аналізу π ланцюжка $\omega \in L(G)$.

$П_0$: Побудуємо корінь дерева та позначимо його аксіомою *S*. Тоді, якщо $\pi = \langle p_1, p_2, ... p_m \rangle$, то

$П_1$: Побудуємо дерево висоти один, взявши зі схеми *P* правило з номером p_1 виду $S \to \alpha_1 \alpha_2 ... \alpha_p$ (Мал. 5):



Мал. 5.

$П_i$: На кроні дерева, отриманого на попередньому кроку, візьмемо перший зліва направо нетермінал (нехай це буде нетермінал α_i) та правило з номером p_i виду: $\alpha_i \to \beta_1 \beta_2 ... \beta_l$ та побудуємо нове дерево. Даний пункт виконувати доти, доки не переглянемо всі елементи з π.

Сформулюємо декілька проблем для стратегії аналізу "зверху донизу":

- у загальному випадку у класі КС-граматик існує проблема неоднозначності (недетермінізму) виводу $\omega \in L(G)$. Як приклад можемо розглянути граматику з "циклами". Це така граматика, у якої в схемі P існує така послідовність правил за участю нетермінала A_i, що:

  $A_i \to A_j$
  $A_j \to A_i$, де A_i — будь-який нетермінал граматики G.

- як наслідок з попереднього пункту, граматики з ліворекурсивним нетерміналом для стратегії аналізу "зверху донизу" недопустимі.
- існують підкласи класу КС-граматик, які природно забезпечують стратегію аналізу "зверху донизу". Один з таких підкласів — це LL(k)-граматики, які забезпечують синтаксичний аналіз ланцюжка $\omega \in L(G)$ за час $O(n)$, де $n = |\omega|$, та при цьому є однозначним.

3.1. Магазинні автомати

Означення. Магазинний автомат M — це сімка $M = \langle Q, \Sigma, \Gamma, q_0, r_0, \sigma, F \rangle$,

де: $Q = \{q_0, q_1, ... q_{m-1}\}$ — множина станів магазинного автомату;

$\Sigma = \{a_1, a_2, ... a_n\}$ — основний алфавіт;

$\Gamma = \{j_1, j_2, ... j_k\}$ — допоміжний алфавіт (алфавіт магазина);

$q_0 \in Q$ — початковий стан магазинного автомату;

$r_0 \in \Gamma^*$ — початковий вміст магазину;

$\sigma : Q * (\Sigma \cup \{\varepsilon\}) * \Gamma \to P(Q * \Gamma^*)$;

$F \subseteq Q$ — множина заключних станів автомата M.

Поточний стан магазинного автомата M описується конфігурацією. Конфігурація магазинного автомата M — це трійка (q, ω, j), де $q \in Q$, $\omega \in \Sigma^*$ $j \in \Gamma^*$. Серед конфігурацій магазинного автомата M виділимо дві:

- початкова конфігурація (q_0, ω, r_0), де $q_0 \in Q$, ω — вхідне слово ($\omega \in \Sigma^*$), $r_0 \in \Gamma^*$;
- заключна конфігурація $(q_f, \varepsilon, \varepsilon)$, $q_f \in F$. В загальній теорії магазинних автоматів іноді як заключну конфігурацію розглядають (q_f, ε, r), де (q_f, r) - фіксована пара. Легко довести, що визначення заключної конфігурації виду $(q_f, \varepsilon, \varepsilon)$ не зменшує потужності класу магазинних автоматів.

Такт роботи (позначається $\models$) магазинного автомата M — це перехід від однієї конфігурації до іншої, а точніше:
$(q_1, a\omega, jr) \models (q_2, \omega, r_1 r)$ при умові, що $(q_2, r_1) \in \sigma(q, a, j)$.

Робота магазинного автомата M (позначається $\models^*$) — це послідовність тактів роботи, а точніше: $(q_1, \omega_1\omega_2, j_1) \models^* (q_2, \omega_2, j_2)$ тоді і тільки тоді, коли $(q_1, \omega_1, j_1) \models (q_1^1, \omega_1^1, j_1^1)$, $(q_1^1, \omega_1^1, j_1^1) \models (q_1^2, \omega_1^2, j_1^2), \ldots, (q_1^{n-1}, \omega_1^{n-1}, j_1^{n-1}) \models (q_2, \omega_2, j_2)$.
Операції $\models$ та $\models^*$ можна трактувати як бінарні відношення на відповідних кортежах. Тоді робота магазинного автомата M — це рефлексивно-транзитивне замикання бінарного відношення $\models$.

Означення. Мова, яку розпізнає магазинний автомат M (позначається $L(M)$) - це множина ланцюжків, які задовольняють умові:

$$L(M) = \{\omega \mid (q_0, \omega, r_0) \models^* (q_0, \varepsilon, \varepsilon), \omega \in \Sigma^*\}.$$

Зафіксуємо наступні результати теорії магазинних автоматів:

1. Не існує алгоритму перетворення недетермінованого магазинного автомата у еквівалентний йому детермінований магазинний автомат.

2. Існує алгоритм, який вирішує проблему порожньої множини $L(M)$ для конкретного магазинного автомата.
3. Існує алгоритм, який за час, пропорційний $O(n^3)$ перевіряє, чи належить $\omega \in \Sigma^*$ мові, яку розпізнає магазинний автомат M.
4. Клас мов, які розпізнаються магазинними автоматами, співпадає з класом мов, що породжуються КС-граматиками.

На основі сформульованих вище результатів для лівосторонньої стратегії виводу $\omega \in \Sigma^*$ в G запропонуємо наступне твердження: для довільної КС-граматики G можна побудувати магазинний автомат M такий, що $L(G) = L(M)$. При цьому автомат буде моделювати лівосторонню стратегію виводу ω в G.

Нехай $G = \langle N, \Sigma, P, S \rangle$ — КС-граматика. Побудуємо відповідний МП-автомат $M = \langle Q, \Sigma, \Gamma, q_0, r_0, \sigma, F \rangle$:

- $Q = \{q_0\}$ — множину станів автомата складає один стан q_0;
- $\Gamma = N \cup \Sigma$ — допоміжний алфавіт;
- $r_0 = S$ — початковий вміст магазина.
- функцію σ визначимо так:
  - якщо $A \to \omega_1 \mid \omega_2 \mid ... \mid \omega_p$ належить P, то
$$\sigma(q_0, \varepsilon, A) = \{(q_0, \omega_1), (q_0, \omega_2), ..., (q_0, \omega_p)\}.$$
  - також поповнимо множину значень функції σ наступними значеннями:
$$\sigma(q_0, a_i, a_i) = \{(q_0, \varepsilon)\}, \quad a_i \in \Sigma.$$

Для ланцюжка $\omega \in \Sigma^*$, $|\omega| = n$ покажемо, якщо ми за m кроків безпосереднього виводу $S \Rightarrow^m \omega$, то відповідний автомат за $(m+n)$ кроків допустить ω. Зробимо перший крок безпосереднього виведення $S \Rightarrow x_1 x_2 ... x_k$ тоді МП-автомат з початкової конфігурації (q_0, ω, S) перейде в наступну конфігурацію $(q_0, \omega, x_1 x_2 ... x_k)$. Далі розглянемо наступні ситуації:

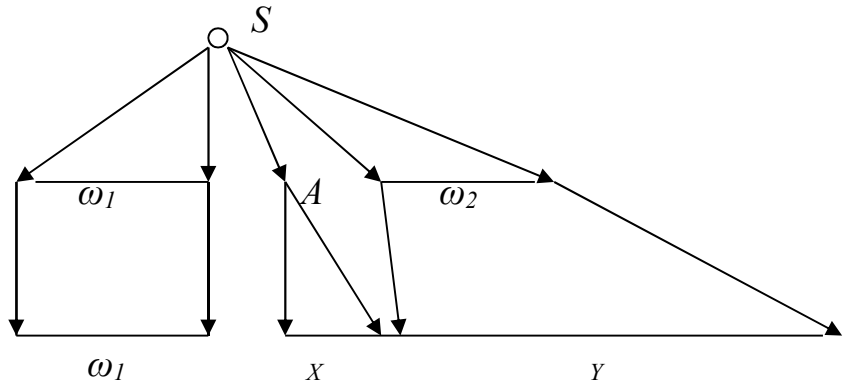
- коли x_1 — термінал a_1 (тобто $\omega = a_1 \omega_1$), тоді МП-автомат виконає наступний такт: $(q_0, a_1\omega_1, a_1 x_2 ... x_k) \models (q_0, \omega_1, x_2 ... x_k)$;
- коли x_1 — нетермінал, тоді в схемі P граматики G виберемо правило виду $x_1 \to y_1 y_2 ... y_l$, зробимо наступний крок безпосереднього виведення : $S \Rightarrow y_1 ... y_l x_2 ... x_k$. При таких умовах автомат перейде в наступну конфігурацію: $(q_0, \omega, x_1 x_2 ... x_k) \models (q_0, \omega, y_1 y_2 ... y_l x_2 ... x_k)$.

Очевидно, якщо ланцюжок ω виводиться за n кроків, то МП-автомат зробить $n+|\omega|$ кроків та розпізнає ω. Таким чином, $L(G) = L(M)$.

3.2. Синтаксичний аналіз без повернення назад

При виводу ланцюжка ω в G на кожному кроку безпосереднього виведення коли ми беремо до уваги виділений нами нетермінал (в залежності від стратегії виведення), виникає питання, яку альтернативу для A_i використати. З точки зору практики, нас цікавить така стратегія виведення ω в граматиці G, коли кожний наступний крок безпосереднього виведення наближав би нас до мети. Ця стратегія

дасть можливість виконати виведення ω в G за час пропорційний $O(n)$, де $n = |\omega|$. Зрозуміло, що не маючи інформації про структуру ω, досягнути вибраної нами мети в більшості випадків неможливо. Але ж тримати інформацію про весь ланцюжок ω також недопустимо. З точки зору практики, отримати потрібний результат розумно при наявності локальної інформації, наприклад, k поточних вхідних лексем програми (k — наперед фіксоване число) достатньо для організації виведення ω в G за час пропорційний $O(n)$. З точки зору синтаксичного аналізу ланцюжка ω мова ведеться про наступну ситуацію:



Мал. 5

Зафіксуємо стратегію виводу: далі будемо розглядати лише лівосторонню стратегію виводу ω в G. Тоді:

- $S \Rightarrow^* \omega_1 A \omega_2$ (A — перший зліва направо нетермінал);
- ω_1 - термінальна частина ланцюжка ω, яку вже виведено (проаналізована частина ланцюжка);
- результат xy, який потрібно ще вивести, виводиться з ланцюжка $A\omega_2$;
- щоб зробити вірний крок виведення (без повернення назад) нам було б достатньо k поточних вхідних символів з непроаналізованої частини програми xy.

Сформульовані нами умови забезпечує клас $LL(K)$-граматик.

Означення. КС-граматика $G = \langle N, \Sigma, P, S \rangle$ називається $LL(k)$-граматикою для деякого фіксованого k, якщо дія двох лівосторонніх виводів виду:

1) $S \Rightarrow^* \omega_1 A \omega_2 \Rightarrow \omega_1 \alpha \omega_2 \Rightarrow^* \omega_1 x$
2) $S \Rightarrow^* \omega_1 A \omega_2 \Rightarrow \omega_1 \beta \omega_2 \Rightarrow^* \omega_2 y$, для яких з $First_k(x) = First_k(y)$ випливає, що $\alpha = \beta$, де $A \to \alpha \mid \beta$.

Неформально, граматика G буде $LL(k)$-граматикою, якщо для ланцюжка $\omega_1 A \omega_2 \in (N \cup \Sigma)^*$ k перших символів (за умови, що вони існують) решти непроаналізованого ланцюжка визначають, що з $A\omega_2$ існує не більше однієї альтернативи виведення ланцюжка, що починається з ω та продовжується наступними k термінальними символами.

Означення.

$$First_k(\alpha) = \{\omega \mid \alpha \Rightarrow^* \omega x, \quad ma \mid \omega \mid = k, \quad тоді\ x \neq \varepsilon \quad або \mid \omega \mid \leq k, \quad тоді\ x = \varepsilon\}$$

Сформулюємо основні твердження стосовно класу $LL(k)$-граматик:

1) Не існує алгоритма, який перевіряє належність КС-граматики класу $LL(k)$-граматик.

2) Існує алгоритм, який для конкретного k перевіряє, чи є задана граматика $LL(k)$-граматикою.
3) Якщо граматика є $LL(k)$-граматикою, то вона є $LL(k+p)$-граматикою, ($p \geq 1$).
4) Клас $LL(k)$-граматик — це підклас КС-граматик, який не покриває його.

Продемонструємо на прикладі справедливість твердження 4. Розглянемо граматику G з наступною схемою P:

$$S \to Sa \mid b$$

Мова, яку породжує наведена вище граматика $L(G) = \{ba^i, i = 0,1,...\}$. Візьмемо виведення наступного ланцюжка $S \Rightarrow^i Sa^i$; за означенням LL(K)-граматики $A = S, \quad \omega_2 = a^i, \quad \alpha = Sa, \quad \beta = b$, тоді $i \geq k$ маємо

$$First_k(Saa^i) \cap First_k(ba^i) = ba^{k-1}$$

Таким чином, КС-граматика G не може бути $LL(k)$-граматикою для жодного k. Як результат, КС-граматика G, яка має ліворекурсивний нетермінал A (нетермінал A називається ліворекурсивний, якщо в граматиці G існує вивід виду $A \Rightarrow^* A\omega$), не може бути $LL(k)$-граматикою.

З практичної точки зору в більшості випадків ми будемо користуватися $LL(1)$-граматиками. У класі $LL(1)$-граматик існує один цікавий підклас - це розподілені $LL(1)$-граматики.

Означення. $LL(1)$-граматика називаються розподіленою, якщо вона задовольняю наступним умовам:

- у схемі P граматики відсутні ε-правила (правила виду $A \to \varepsilon$);
- для нетермінала A праві частини A-правила починаються різними терміналами.

Для подальшого аналізу означення $LL(k)$-граматики розглянемо алгоритм обчислення функції $First_k(\alpha) \quad \alpha \in (N \cup \Sigma)$.

Означення: Якщо $\alpha = \omega_1 \cdot \omega_2$, то $First_k(\omega_1 \cdot \omega_2) = First_k(\omega_1) \oplus_k First_k(\omega_2)$, де $\oplus_k$ — бінарна операція над словарними множинами (мовами):

$$L_1 \oplus_k L_2 = \{\omega \mid \omega\omega_1 = xy, \quad де\ x \in L_1, y \in L_2, \quad |\omega| = k, якщо\ \omega_1 \neq \varepsilon; \quad |\omega| \leq k, якщо\ \omega_1 = \varepsilon\}.$$

Висновок, якщо $\omega = \alpha_1\alpha_2...\alpha_p$, де $\alpha_i \in (N \cup \Sigma)$, тоді

$$First_k(\alpha_1\alpha_2...\alpha_p) = First_k(\alpha_1) \oplus_k First_k(\alpha_2) \oplus_k ... \oplus_k First_k(\alpha_p)$$

Очевидно, якщо $\alpha_i \in \Sigma$, то $First_k(\alpha_i) = \{\alpha_i\}$ при $k > 0$. Розглянемо алгоритм пошуку $First_k(A_i), \quad A_i \in N$.

Алгоритм. Пошук множини $First_k(A_i), \quad A_i \in N$.

Визначимо значення функції $F_i(x)$ для кожного $x \in (N \cup \Sigma)$.

Π_1. $F_i(a) = \{a\}$ для всіх $a \in \Sigma, \quad i \geq 0$.

Π_2. $F_0(A_i) = \{\omega \mid \omega \in \Sigma^{*k} \quad для\ A_i \to \omega x \quad при\ (|\omega| = k \wedge x \neq \varepsilon) \vee (|\omega| \leq k \wedge x = \varepsilon)\}$

в інших випадках - *невизначено*.

Π_n.

$$F_n(A_i) = F_{n-1}(A_i) \cup \{\omega \mid \omega \in \Sigma^{*k}, \omega \in F_{n-1}(\alpha_1) \oplus_k ... \oplus_k F_{n-1}(\alpha_p)$$
$$для\ правила\ A_i \to \alpha_1...\alpha_p, \quad за\ умови, що \quad F_{n-1}(\alpha_1), F_{n-1}(\alpha_2), ...F_{n-1}(\alpha_p)\ визначені\}$$

в інших випадках - *невизначено.*

Π_m. $F_m(A_i) = F_{m+1}(A_i) = ...$ для всіх $A_i \in N$.

Очевидно, що:

- послідовність $F_0(A_i) \subseteq F_1(A_i) \subseteq \ldots$ — монотонно зростаюча;
- $A_n(A_i) \subseteq \Sigma^{*k}$ — послідовність, обмежена зверху.

Тоді покладемо $First_k(A_i) = F_m(A_i)$ для кожного $A_i \in N$.

Приклад: Знайти множину $First_k(A_i)$ для нетерміналів граматики з наступною схемою правил:

S -> BA
A -> +BA | ε
B -> DC
C -> *DC | ε
D -> (S) | a

Нехай k=2.

$F_n \backslash A_i$	S	A	B	C	D
F_0	--	{ ε }	--	{ ε }	{a}
F_1	--	{ ε }	{a}	{ ε, *a}	{a}
F_2	{a}	{ε, +a}	{a, a*}	{ ε, *a}	{a}
F_3	{a, a+, a*}	{ε, +a}	{a, a*}	{ ε, *a}	{a, (a}
F_4	{a, a+, a*}	{ε, +a}	{a, a*, (a}	{ ε, *a, *(}	{a, (a}
F_5	{a, a+, a*, (a}	{ε, +a, +(}	{a, a*, (a}	{ ε, *a, *(}	{a, (a}
F_6	{a, a+, a*, (a}	{ε, +a, +(}	{a, a*, (a}	{ ε, *a, *(}	{a, (a, ((}
F_7	{a, a+, a*, (a}	{ε, +a, +(}	{a, a*, (a, ((}	{ ε, *a, *(}	{a, (a, ((}
F_8	{a, a+, a*, (a, ((}	{ε, +a, +(}	{a, a*,(a, ((}	{ ε, *a, *(}	{a, (a, ((}
F_9	{a, a+, a*, (a, ((}	{ε, +a, +(}	{a, a*, (a, ((}	{ ε, *a, *(}	{a, (a, ((}

Скористаємося означенням $First_k(\alpha)$ сформулюємо необхідні й достатні умови, за яких КС-граматика буде $LL(k)$-граматикою:

для довільного виводу в граматиці G виду $S \Rightarrow_L^* \omega_1 A \omega_2$ та правила $A \to \alpha \mid \beta$:

$$First_k(\alpha \cdot \omega_2) \cap First_k(\beta \cdot \omega_2) = \varnothing \qquad (1)$$

Вище сформульована умова для $LL(k)$-граматик може бути перефразована з урахуванням визначення множини $First_k$:

для довільного виводу в граматиці G виду $S \Rightarrow^* \omega_1 A \omega_2$ та правила $A \to \alpha \mid \beta$:

$$First_k(\alpha \cdot L) \cap First_k(\beta \cdot L) = \varnothing, \text{ де } L = First_k(\omega_2) \qquad (2).$$

Оскільки $L \subseteq \Sigma^{*k}$, то умова (2) є конструктивною умовою і може бути використана для перевірки, чи КС-граматика є $LL(k)$-граматикою для фіксованого k.

Означення: КС-граматика називається сильною $LL(k)$-граматикою, якщо для A-правила виду $A \to \alpha \mid \beta$ задовольняється умова:

$$First_k(\alpha \cdot Follow_k(A)) \cap First_k(\beta \cdot Follow_k(A)) = \varnothing,$$

де $Follow_k(\alpha)$ $\alpha \in (N \cup \Sigma)^*$ визначається так:

$$Follow_k(\alpha) = \{\omega \mid S \Rightarrow^* \omega_1 \alpha \omega_2, \omega \in First_k(\omega_2)\}$$

Операції $First_k$ та $Follow_k$ можна узагальнити для словарної множини L, тоді:

$$First_k(L) = \{\omega \mid \omega \in First_k(\alpha_i)\ \forall \alpha_i \in L\}$$

$$Follow_k(L) = \{\omega \mid S \Rightarrow^* \omega_1 \alpha_i \omega_2\ \omega \in First_k(\omega_2)\ \forall \alpha_i \in L\}.$$

Без доведення зафіксуємо наступні твердження:

- кожна $LL(1)$-граматика є сильною $LL(1)$-граматикою;
- існують $LL(k)$-граматики (k>1), які не є сильними $LL(k)$-граматиками.

На прикладі продемонструємо останнє твердження. Нехай граматика G визначена наступними правилами:

$$S \to aAaa \mid bAba\ ,\ A \to b \mid \varepsilon\ .$$

Відповідні множини $First_2(S) = \{ab, aa, bb\}$, $First_2(A) = \{b, \varepsilon\}$, $Follow_2(A) = \{aa, ba\}$, $Follow_2(S) = \{\varepsilon\}$.

Перевіримо умову для сильної $LL(2)$-граматики:

а) виконаємо перевірку $LL(2)$-умови для правила $S \to aAaa \mid bAba$

$$First_2(aAaa \cdot Follow_2(S)) \cap First_2(bAba \cdot Follow_2(S)) =$$
$$= (First_2(aAaa) \oplus_2 Follow_2(S)) \cap (First_2(bAba) \oplus_2 Follow_2(S)) =$$
$$= (\{ab, aa\} \oplus_2 \{\varepsilon\}) \cap (\{bb\} \oplus_2 \{\varepsilon\}) = \{ab, aa\} \cap \{bb\} = \varnothing$$

б) виконаємо перевірку $LL(2)$-умови для правила $A \to b \mid \varepsilon$

$$First_2(b \cdot Follow_2(A)) \cap First_2(\varepsilon \cdot Follow_2(A)) =$$
$$= \{ba, bb\} \cap \{aa, ba\} = \{ba\}$$

Висновок: вище наведена граматика не є сильною $LL(2)$-граматикою. Перевіримо цю ж граматику на властивість $LL(2)$-граматики. Тут ми маємо два різні варіанти виводу з S:

а) $S \Rightarrow^* aAaa$

$$First_2(b \cdot aa) \cap First_2(\varepsilon \cdot aa) = \{ba\} \cap \{aa\} = \varnothing$$

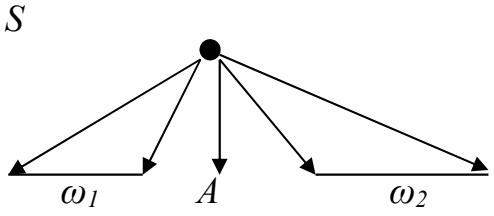
б) $S \Rightarrow^* bAba$

$$First_2(b \cdot ba) \cap First_2(\varepsilon \cdot ba) = \{bb\} \cap \{ba\} = \varnothing$$

Висновок: наведена вище граматика є $LL(2)$-граматикою.

Алгоритм. Обчислення $Follow_k(A_i),\quad A_i \in N$.

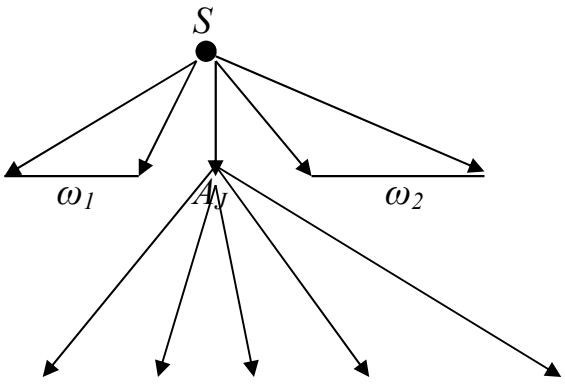
Для побудови алгоритму пошуку множини $Follow_k(A_i),\quad A_i \in N$ розглянемо наступні приклади на синтаксичному дереві, які дозволять перейти до узагальнень. Подивимося на синтаксичне дерево висоти 1 для правила $S \to \omega_1 A \omega_2$



Мал. 6

Тоді $First_k(\omega_2) \subseteq Follow_k(A)$.

Далі розглянемо дерево висоти 2:



Мал. 7.

Тоді $First_k(\omega_4\omega_2) \subseteq Follow_k(A)$. В силу вищесказаного, будемо знаходити значення функції $\sigma(S, A_i)$, тобто будемо розглядати всілякі дерева, які можна побудувати, починаючи з аксіоми S.

$П_0$. $\sigma_0(S,S) = \{\varepsilon\}$. Очевидно, за 0 кроків ми виведемо *S*, після якої знаходиться ε. У інших випадках $\sigma_0(S, A_i)$ — невизначено $A_i \in N \setminus \{S\}$.

$П_1$. $\sigma_1(S, A_i) = \sigma_0(S, A_i) \cup \{\omega \mid S \to \omega_1 A_i \omega_2, \omega \in First_k(\omega_2)\}$. В інших випадках $\sigma_1(S, A_i)$ — невизначено.

….

$П_n$ $\sigma_n(S, A_i) = \sigma_{n-1}(S, A_i) \cup \{\omega \mid A_j \to \omega_1 A_i \omega_2, \omega \in First_k(\omega_2 \cdot \sigma_{n-1}(S, A_j))\}$. В інших випадках $\sigma n(S, A_i)$ — невизначено.

….

Настане крок $П_m$, коли $\sigma_m(S, A_i) = S_{m+1}(S, A_i) = \ldots$, $\forall A_i \in N$.

Тоді покладемо $Follow_k(A_i) = \sigma_m(S, A_i)$ для кожного $A_i \in N$.

Очевидно, що:

- послідовність $\sigma_0(S, A_i) \subseteq \sigma_1(S, A_i) \subseteq \ldots$ монотонно зростаюча;
- $\sigma_n(S, A_i) \subseteq \Sigma^{*k}$ послідовність обмежена зверху.

До того, як перевірити граматику на $LL(k)$-властивість необхідно перевірити її на наявність ліворекурсивних нетерміналів та спробувати уникнути лівої рекурсії.

Означення: Нетермінал A_i КС-граматики G називається ε-нетерміналом, якщо $A_i \Rightarrow^* \varepsilon$.

Алгоритм. Пошук ε-нетерміналів:

$S_0 = \{A_i \mid A_i \to \varepsilon\}$

$S_1 = S_0 \cup \{A_i \to \alpha_1\alpha_2\ldots\alpha_p, \alpha_j \in S_0 \ j = 1..p\}$

….

$S_n = S_{n-1} \cup \{A_i \to \alpha_1\alpha_2\ldots\alpha_p, \alpha_j \in S_{n-1} \ j = 1..p\}$

…. $П_n$

$S_m = S_{m+1} = \ldots$

Тоді множина S_m — множина ε-нетерміналів.

Приклад. Для граматики G з схемою правил P знайдемо множину ε-нетерміналів:

S -> a B D | D | A C |b

A -> S C B | S A B C |C b D | ε

B -> C A | d

C -> A D C |a | ε

D -> E a C |S C

E -> B C S |a

$S_0 = \{A, C\}$

$S_1 = \{A, C\} \cup \{B, S\}$

$S_2 = \{A, B, C, S\} \cup \{D\}$

$S_3 = \{S, A, B, C, D\} \cup \{E\}$

$S_4 = \{S, A, B, C, D\} \cup \{E\}$

Таким чином, множина ε-нетерміналів для наведеної вище граматики - $\{S, A, B, C, D, E\}$.

Алгоритм. Тестування нетермінала A_i на ліву рекурсію. Для кожного нетермінала A_i побудуємо наступну послідовність множин S_0, S_1, ….

$S_0 = \{A_i \mid A_j \to \omega_1 A_j \omega_2, \omega_1 \Rightarrow^* \varepsilon\}$, починаємо з нетерміналу A_i.

$S_1 = S_0 \cup \{A_j \mid A_l \to \omega_1 A_j \omega_2, \quad \omega_1 \Rightarrow^* \varepsilon, \quad A_l \in S_0\}$

….

$S_n = S_{n-1} \cup \{A_j \mid A_l \to \omega_1 A_j \omega_2, \quad \omega_1 \Rightarrow^* \varepsilon, \quad A_l \in S_{n-1}\}$

….

$S_m = S_{m+1} = ...$

Тоді якщо $A_i \in S_m$, то A_i — ліворекурсивний нетермінал.

Приклад. Для граматики G з схемою правил P знайдемо множину ліворекурсивних нетерміналів:

S - > A b S | A C

A - > B D

C - > S a | ε

B - > B C | ε

D - > a B | B A

1. Виконаємо процедуру тестування для кожного нетермінала окремо:

- наприклад, для нетермінала *S*:

$S_0 = \{A\}$

$S_1 = \{A, B, D\}$

$S_2 = \{A, B, D, C\}$

$S_3 = \{A, B, D, C, S\}$

Запропонуємо декілька прийомів, що дають можливість при побудові $LL(k)$ – граматик уникнути лівої рекурсії. Розглянемо граматику зі схемою правил $S \to Sa \mid b$, яка має ліворекурсивний нетермінал S. Замінимо схему правил новою схемою з трьома правилами

$S \to bS_1$

$S_1 \to aS_1 \mid \varepsilon$.

Приклад: Для граматики G з схемою правил P для кожного нетермінала знайдемо множину $Follow_1(A)$ (k=1):

S -> BA

A -> +BA | ε

B -> DC

C -> *DC | ε

D -> (S) | *a*

З прикладу, що наведено раніше множини $First_1(A)$,будуть такими:

$First_1(S)= First_1(B)= First_1(D)=\{(,a\}$, $First_1(A)=\{+, \varepsilon\}$, $First_1(C)=\{*, \varepsilon\}$.

$\delta_n \backslash A_i$	S	**A**	**B**	**C**	**D**
δ_0	{ ε }	--	--	--	--
δ_1	{ ε }	{ ε }	{+, ε }	--	--
δ_2	{ ε }	{ε}	{+, ε }	{+, ε }	
δ_3	{ ε }	{ε}	{+, ε }	{+, ε }	{*, +, ε }
δ_4	{ ε,) }	{ε}	{+, ε }	{+, ε }	{*, +, ε }
δ_5	{ ε,) }	{ε,)}	{+, ε }	{+, ε,)}	{*, +, ε }
δ_6	{ ε,) }	{ε,)}	{+, ε ,)}	{+, ε }	{*, +, ε ,)}
δ_7	{ ε,) }	{ε,)}	{+, ε ,)}	{+, ε ,)}	{*, +, ε ,)}

Таким чином, $Follow_1(S)=$ { ε,) }, $Follow_1(A)=$ {ε,)}, $Follow_1(B)=$ {+, ε,)}, $Follow_1(C)=$ {+, ε ,)}, $Follow_1(D)=$ {*, +, ε ,)}.

3.3. Синтаксичний аналіз на основі $LL(1)$-граматик

Скориставшись означенням $LL(1)$-граматики, сформулюємо умови для $LL(1)$-граматики: граматика G буде $LL(1)$-граматикою тоді і тільки тоді, коли кожного А-правила виду $A \to \omega_1 \mid \omega_2 \mid ... \mid \omega_p$

- $First_1(\omega_i) \cap First_1(\omega_1) = O, i \neq j$
- якщо $\omega_i \Rightarrow^* \varepsilon$, $First_1(\omega_j) \cap Follow_1(A) = \varnothing$, $1 \le j \le p, j \neq i$

Означення. Таблиця $M(a,b)$, $a \in N \cup \Sigma \cup \{\varepsilon\}$, $b \in \Sigma \cup \{\varepsilon\}$ управління LL(1)-синтаксичним аналізатором визначається таким чином:

1. $M(A,b)$ — це номер правила виду $A \to \omega_i$ такого, що $b \in First_1(\omega_i \cdot Follow_1(A))$
2. $M(a,a)$ — "виштовхнути" для всіх $a \in \Sigma$
3. $M(\varepsilon,\varepsilon)$ — "допустити"
4. в інших випадках $M(a,b)$ — невизначено.

Побудуємо таблицю управління для наступної граматики:

$S \to BA$	(1)	$First_1(S) = \{(,a\}$	$First_1(A) = \{+,\varepsilon\}$
$A \to +BA$	(2)	$First_1(B) = \{(,a\}$	$First_1(C) = \{*,\varepsilon\}$
$A \to \varepsilon$	(3)	$First_1(D) = \{(,a\}$	
$B \to DC$	(4)		
$C \to *DC$	(5)	$Follow_1(A) = \{\varepsilon,)\}$	$Follow_1(B) = \{+,\varepsilon,)\}$
$C \to \varepsilon$	(6)	$Follow_1(C) = \{+,\varepsilon,)\}$	$Follow_1(D) = \{+,*,\varepsilon,)\}$
$D \to (S)$	(7)	$Follow_1(S) = \{\varepsilon,)\}$	
$D \to a$	(8)		

Знайдемо множини $First_1(\omega) \oplus_1 Follow_1(A)$, $A \to \omega$.

Правило	Номер правила	$First_1(\omega) \oplus_1 Follow_1(A)$

$S \to BA$	(1)	$\{(, a\}$
$A \to +BA$	(2)	$\{+\}$
$A \to \varepsilon$	(3)	$\{\varepsilon,)\}$
$B \to DC$	(4)	$\{(, a\}$
$C \to *DC$	(5)	$\{*\}$
$C \to \varepsilon$	(6)	$\{+, \varepsilon,)\}$
$D \to (S)$	(7)	$\{(\}$
$D \to a$	(8)	$\{a\}$

При побудові таблиці $M(a,b)$ управління $LL(1)$-синтаксичним аналізатором достатньо лише побудувати першу її частину, тобто $M(A_i,b)$, оскільки "діагональ" таблиці $M(a,a), a \in \Sigma$ та $M(\varepsilon,\varepsilon)$ визначаються стандартно.

A_i\Σ	**a**	**(**	**)**	+	*	ε
S	1	1				
A			3	2		3
B	4	4				
C			6	6	5	6
D	8	7				

Алгоритм. Побудова $LL(1)$- синтаксичного аналізатора на основі таблиці управління $M(a,b)$**:**

$П_0$ Прочитаємо поточну лексему з вхідного файла, у стек магазинного автомата занесемо аксіому S.

….

$П_i$ - Якщо на вершині стека знаходиться нетермінал A_i, то активізувати рядок таблиці, позначений A_i. Елемент $M(A_i, \langle поточна\ лексема \rangle)$ визначає номер правила, права частина якого заміняє A_i на вершині стека.

- Якщо на вершині стека лексема $a_i = \langle поточна\ лексема \rangle$, то з вершини стека зняти a_i та прочитати нову поточну лексему.
- Якщо стек порожній та досягли кінця вхідного файла, то вхідна програма синтаксично вірна.
- В інших випадках — синтаксична помилка.

У деяких випадках досить складно (а інколи й принципово неможливо побудувати $LL(1)$-граматику для реальної мови програмування. При цьому $LL(1)$-властивість задовольняється майже для всіх правил - лише декілька правил створюють конфлікт, але для цих правил задовольняється сильна $LL(2)$-властивість. Тоді таблиця $M(a,b)$ визначається в такий спосіб:

- $M(A,b) = \langle номер\ правила \rangle$ виду $A \to \omega_i$, такого, що $b \in First_1(\omega_1 \cdot Follow_1(A))$
- $M(A,b) = \langle ім'я\ програми \rangle$ за умови, що

$$b \in First_1(\omega_i \cdot Follow_1(A)) \cap First_1(\omega_j \cdot Follow_1(A)), \quad i \neq j .$$

Програма, яка виконує додатковий аналіз вхідного ланцюжка, повинна:

- прочитати додатково одну лексему;
- на основі двох вхідних лексем вибрати необхідне правило або сигналізувати про синтаксичну помилку;
- у випадку, коли правило вибрано, необхідно повернути додатково прочитану лексему у вхідний файл.

Звичайно, необхідно модифікувати алгоритм LL(1)-синтаксичного аналізатора. При цьому підпрограма аналізу конфліктної ситуації повинна додатково прочитати нову вхідну лексему, далі скориставшись контекстом з двох лексем, визначити номер правила, яке замість нетермінала на вершині стека та повернути додатково прочитану лексему у вхідний файл.

3.4. LL(1)-синтаксичний аналізатор для мови Pascal

Наведемо текст LL(1)-синтаксичного аналізатора для мови програмування Pascal, зробивши деякі пояснення:

- Синтаксичний аналізатор мови програмування Pascal використовує для виділення лексем з текстового файла функцію pascal_scaner(), яка при кожному звернені до неї виділяє з файла програми нову лексему. Коли сканер досягне кінця файла, то в подальшому EOF передається як нова лексема. Текст виділеної лексеми знаходиться в змінній lexema[], яка є зовнішньою змінною.
- Допоміжна функція index_elem () знаходить індекс (порядковий номер починаючи з нуля) у відповідному масиві терміналів або нетерміналів.

```
#include  <stdio.h>
#include  "mystand.h"
/* визначення структури стека синтаксичого аналізатора  */
#define MAX_STACK 200
int STACK[MAX_STACK], POS_STACK=0;
#define NULL_STACK() (POS_STACK? 0 : 1)
#define COPY_STACK() (STACK[POS_STACK])
#define PUSH_STACK() (POS_STACK? --POS_STACK :0)
#define DOWN_STACK(c) (POS_STACK < MAX_STACK ? STACK[++POS_STACK]=c,1
: 0)
extern int  pascal_scaner(void);     // функція виділення нової лексеми
extern void scaner_close(void);      // функція, що закриває вхідний файл
extern char lexema[];                // поточна вхідна лексема
extern int lexema_code;              // код виділеної лексеми
extern int lexema_line;              // рядок, з якого прочитана лексема
extern int lexema_pos;               // позиція лексеми у рядку
extern int  index_elem(int *, int, int);
// Зовнішні змінні для синтаксичного аналізатора:
//        - таблиця управління LL(1) - синтаксичним аналізатором - TABL_LL1_UPR.
//          Кількість рядків таблиці numnet - кількість нетерміналів граматики,
//        кількість стовпчиків - (numtrm+1) - кількість терміналів в граматиці
//  Допоміжна функція, яка визначає індекс термінала або нетермінала у
//   відповідному масиві.
int index_elem(int *net_term, int num, int elem)
        {  int i;
```

```
                for (i=0; i < num; i++) if (*(net_term+i) = = elem) return i;
                }
// Лексичний аналізатор:
//        - виділяє лексему в поле lexema,
//        - в поле lexema_code заносить код лексеми.
//        - аналізатор повертає Е-епсилон слово(lexema_code==0), коли досягли EOF,
//                        інакше код лексеми в полі lexema_code
// В lexema_line знаходиться номер рядка, з якого прочитана лексема.
// В lexema_pos знаходиться позиція в рядку, з якого прочитана лексема.

int ll1_parser_pascal(q,r)
   struct node *q;
   struct dnode *r;
  { struct node *qw; int i,  line0, colomn, ind;
   int upr;
   /* початкові установки для синтаксичного аналізатора */
   STACK[0]=0; STACK[1]=*(q->pd); POS_STACK=1; ind=0; lexema_code=0;
   /* головний цикл работи синтаксичного аналізатора */
   while (! NULL_STACK())  // поки стек не пустий
   { if (!ind) ind=1, pascal_scaner();
// A. Обробка при умові, що на вершині стека термінал
        if (COPY_STACK() >= 0)
            { if (COPY_STACK() == lexema_code)
                { ind=0; PUSH_STACK();   continue; }
        //  можливо короткий if
            if (strcmp(NAME_ELEM(COPY_STACK()),"else") == 0)
                { PUSH_STACK();  PUSH_STACK(); continue; } // короткий if
        // Синтаксична помилка
         printf("Синтаксична помилка: рядок - %5.5i, позиція -
                        %3.3i\n",lexema_line,lexema_pos);
         printf("Пропущена лексема %s\n",NAME_ELEM(COPY_STACK()));
         printf("Вершина стека -  %s, вхідна лексема - %s -
          %s\n",NAME_ELEM(COPY_STACK()),lexema,NAME_ELEM(lexema_code));
         scaner_close();  return(0);
         }
// B. Обробка при умові, що на вершині нетермінал
          line0=index_elem(netname,numnet,COPY_STACK());
          if (lexema_code) colomn=index_elem(terminal,numtrm,lexema_code);
                        else colomn=numtrm;
        if  (upr= *(TABL_LL1_UPR+line0*(numtrm+1)+colomn))
                {    PUSH_STACK();
        // пошук продукції в списку продукцій
         for(qw=q,i=1;  i < upr; i++,qw=qw->next)  ;
        // запис правої частини продукції в стек
         for ( i=qw->len-1;  i >  0;  i-- )     DOWN_STACK(*(qw->pd+i));
         continue;
        }
// Синтаксична помилка в програмі
        printf("Синтаксична помилка: рядок - %5.5i, позиція -
```

```
                %3.3i\n",lexema_line,lexema_pos);
          printf("Вершина стека -  %s, вхідна лексема - %s -
         %s\n",NAME_ELEM(COPY_STACK()),lexema,NAME_ELEM(lexema_code));
          scaner_close(); return(0);
         }  // кінець цикла обробки  стека
//  Стек порожній: перевіримо стан справ на вході
         if (! ind ) pascal_scaner();   scaner_close();
         if (lexema_code == 0) {
           printf("\nВ програмі синтаксичних помилок немає\n");return(1);
           }
          else {
           printf("\nЛогічний кінець програми знайдено до кінця вхідного файла\n");
           return(0);
           }
  }   //  кінець програми
```

3.5. Метод рекурсивного спуску програмування синтаксичних аналізаторів

Означення. Синтаксична діаграма — це орієнтований граф, дуги котрого позначені елементами $(N \cup \Sigma)^*$. Синтаксична діаграма будується для кожного A-правила КС-граматики мови програмування.

Оскільки вершини такого графа не іменуються, то вони припускаються неявно. Синтаксична діаграма позначається іменем нетермінала, для якого вона будується.

Мета побудови синтаксичних діаграм для мови програмування на основі КС-граматики:

- для кожного A-правила КС-граматики будується синтаксична діаграма;
- на основі побудови синтаксичної діаграми для деякого нетермінала $A \in N$ будуємо підпрограму, яка аналізує ту частину головної програми, яку вона визначає.

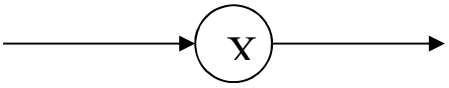
Оскільки у більшості випадків при визначенні синтаксису мови програмування ми користуємося множиною рекурсивних правил, то серед підпрограм, які будуються на основі правил граматики, будуть і рекурсивні процедури (рекурсія буде як явна, так і неявна).

Сформулюємо правила побудови синтаксичного графа:

$П_1$. Кожен нетермінал з відповідною множиною породжуючих правил $A \to \omega_1 \mid \omega_2 \mid ... \mid \omega_p, \omega_i \in (N \cup \Sigma)^*, i = 1...p$ відображається в один синтаксичний граф. Отже, кількість синтаксичних графів рівна кількості нетерміналів граматики G.

$П_2$. Для кожного елемента ланцюжка $\omega = \alpha_1\alpha_2...\alpha_p, \alpha_i \in N \cup \Sigma, i = 1...p$ будуємо ребро синтаксичного графа та покажемо його таким чином що:

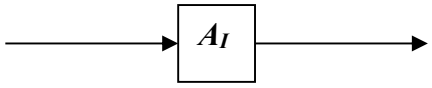
- якщо $\alpha_i = x, x \in \Sigma$, де x — вихідна лексема, то будуємо таке ребро



(1)

Мал.1.

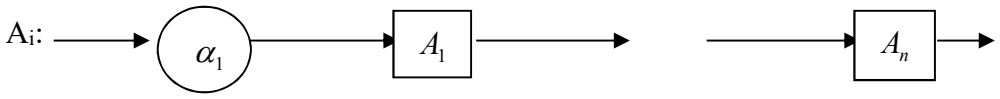
- якщо $\alpha_k = A_i \in N$ — нетермінал граматики , то будуємо таке ребро



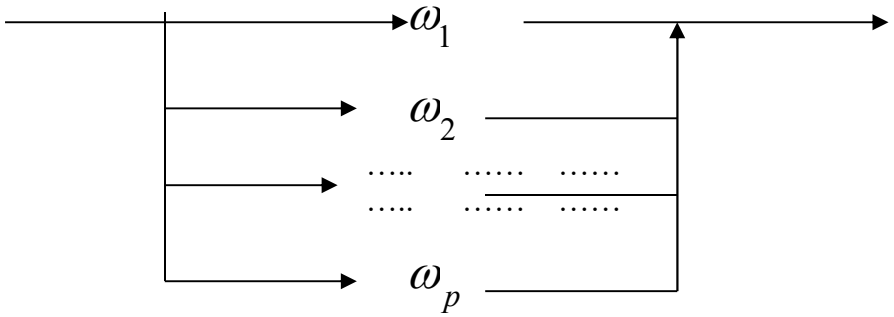
(2)

Мал.2 .

Тоді, коли правило граматики G має вигляд $A_i \to \alpha_1 A_1 ... \alpha_p A_p$ для побудови діаграми скористаємося способами (1) та (2):



Коли правило граматики G має вигляд $A_i \to \omega_1 | \omega_2 | ... | \omega_p, \quad \omega_i \in (N \cup \Sigma)^*, i = 1...p$, то відповідний синтаксичний граф буде мати вигляд:



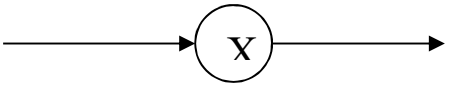
Мал. 3.

де замість $\omega_1, \omega_2, ... \omega_p$ будуються відповідні синтаксичні діаграми.

Якщо на основі граматики мови програмування побудована множина синтаксичних графів, то можна спробувати зменшити їх кількість, скориставшись підстановкою одних графів у інші. При цьому замість елемента A_i підставляється його синтаксичний граф. Таким чином можна зменшити кількість синтаксичних графів. Для того, щоб забезпечити детермінований синтаксичний аналіз з переглядом вперед на одну лексему, потрібно накласти певні обмеження, а саме:

- для кожного правила A_i виду $A_i \to \omega_1 | \omega_2 | ... | \omega_p$ з синтаксичною діаграмою виду (3) необхідне виконання наступної умови: множини $First_1(\omega_j) \oplus_1 Follow_1(A_i), \ j = 1...p$ повинні попарно не перетинатися. Зрозуміло, що ця умова гарантує детермінований вибір шляху при русі по синтаксичній діаграмі. Подальше програмування синтаксичного аналізатора можна звести до наступних примітивів:

- для фрагмента синтаксичної діаграми виду



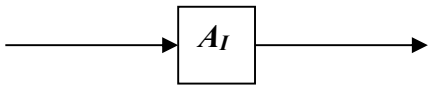
Мал.4.

відповідний фрагмент програми (наприклад, мовою С) матиме вигляд:

extern int lexema_code;// код лексеми, яку виділив сканер
extern char lexema_text[ ];// текст лексеми

…
if (lexema_code==code_x) get_lexema( );
else error( );

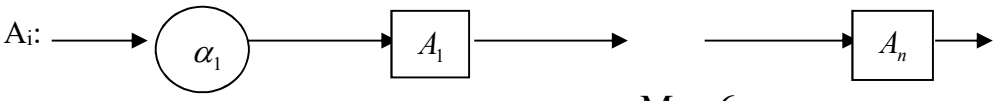
- для фрагмента синтаксичної діаграми виду



Мал.5.

відповідний фрагмент програми матиме вигляд:
//виклик функції, яка побудована для синтаксичної діаграми
f_Ai( );
// побудованої для нетермінала A_i.

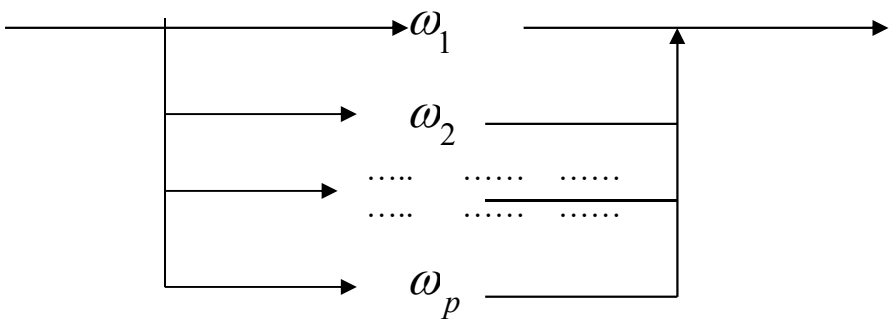
- Для послідовності елементів синтаксичної діаграми виду



Мал.6.

відповідний фрагмент програми матиме вигляд:
extern int lexema_code;
extern char lexema_text[ ];
…
{if (lexema_code==code_x1) get_lexema( );
else error( );
f_A1( );
…
if (lexema_code==code_xn) get_lexema( );
else error( );
}

Для фрагменту синтаксичної діаграми, побудованої для А-правила, яка має вигляд:



Мал.7.

для кожного $\omega_i, i = 1...p$ знайдемо такі множини:

- $First_1(\omega_1) \oplus_1 Follow_1(A)$ для ω_1;
- $First_1(\omega_2) \oplus_1 Follow_1(A)$ для ω_2;

…

- $First_1(\omega_{p1}) \oplus_1 Follow_1(A)$ для ω_p.

Нехай ми знайшли відповідні множини для кожного $\omega_i, i = 1...p$:

- $L_1 = \{a_1^1, a_1^2, ..., a_1^{n_1}\}$ для ω_1;
- $L_2 = \{a_2^1, a_2^2, ..., a_2^{n_2}\}$ для ω_2;

…

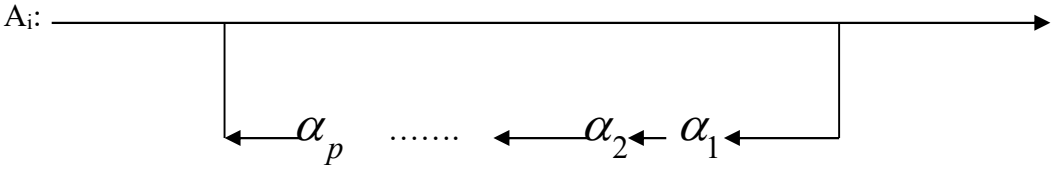
- $L_p = \{a_p^1, a_p^2, ..., a_p^{n_p}\}$ для ω_p.

Оскільки за умовою $L_i \cap L_j = \varnothing, i \neq j$, то відповідний фрагмент програми на мові С матиме вигляд:

```
extern int lexema_code;
extern char lexema_text[ ];
….    ….
void Ai (void)
{switch(lexema_code)
        {case code_a_1^1 :
         case code_a_1^2 :
        …
         case code_a_1^{n_1} :// фрагмент програми для ω_1
        …
        break;
        case code_a_2^1 :
         case code_a_2^2 :
        …
         case code_a_2^{n_2} :// фрагмент програми для ω_2
        …
        break;
        …    ….    ….
        case code_a_p^1 :
         case code_a_p^2 :
        …
         case code_a_p^{n_p} :// фрагмент програми для ω_p
        …
        break;
        default: error( );
        }
        }//кінець функції для нетермінала A_i
```

Відмітимо, що до того, як зменшувати кількість синтаксичних діаграм шляхом суперпозиції одних діаграм в інші, необхідно знайти контексти виду $First_1(\omega_i) \oplus_1 Follow_1(A)$ для тієї синтаксичної діаграми нетермінала A, для якої ми виконуємо операцію суперпозиції. Ці контексти ми використаємо при програмуванні синтаксичного аналізатора на основі синтаксичної діаграми, у яку підставлено синтаксичну діаграму для нетермінала A.

Досить часто при визначенні синтаксису мови програмування користуються синтаксичними правилами виду $A_i \to \alpha_1\alpha_2...\alpha_p A_i \mid \varepsilon$. Тоді синтаксична діаграма буде мати вигляд:



Мал. 8 .

Для вище наведеної синтаксичної діаграми відповідні множини будуть:

- $L_1 = First_1(\alpha_1\alpha_2...\alpha_p A_i) \oplus_1 Follow_1(A_i)$ для $\alpha_1\alpha_2...\alpha_p A_i$;
- $L_2 = Follow_1(A_i)$ для непоміченого ребра.

Відповідний фрагмент програми мовою С матиме вигляд:

```
extern int lexema_code;
extern char lexema_text[ ];
void Ai(void)
{ while (lexema_code= =code_ x_1^1 ||
        lexema_code= =code_ x_1^2 ||
        ...
        lexema_code= =code_ x_1^{n_1} )
        {// фрагмент програми для ланцюжка α1α2...αp
        ...
              }
}// кінець підпрограми для нетермінала Ai
```

Виконавши аналіз варіантів побудови синтаксичного аналізатора на основі синтаксичних діаграм, покажемо вигляд основної — main-програми:

```
int lexema_code;
char lexema_text [500];
int main ( )
{ get_lexema ( );
  Axioma_S ( );//процедура, пов'язана з аксіомою граматики
}
```

3.6. Побудова LL(k)-синтаксичного аналізатора (k>1).

Повернемось до умови, при якій граматика *G* буде LL(k)-граматикою, а саме: для довільного виводу:

$$S \Rightarrow^* \varpi_1 A \varpi_2 \quad \text{та правила } A \to \alpha \mid \beta$$

$$First_k(\alpha\ L) \quad \cap \quad First_k(\beta\ L) = O, \quad де \quad L = First_k(\varpi_2)$$

Оскільки $L \subseteq \Sigma^{*k}$ - конструктивна множина, спробуємо побудувати всілякі множини L, які задовольняють попередньо сформульованій умові.

Означення. Множина

$$Local_k(S, A) = \{L \mid S \Rightarrow *xAw, \quad L = First_k(w) \quad при\ відповідних\ x\ та\ w\}$$

Алгоритм. Пошук множини $Local_k(S, A)$:

Π_0: $\delta_0(S, S) \quad = \quad \{\{\varepsilon\}\}$

в інших випадках - невизначено.

Π_1: $\delta_1(S, A_i) \quad = \quad \delta_0(S, A_i) \bigcup \{L \mid S \to \varpi_1 A_i \varpi_2, \quad L = First_k(w_2)\}$

в інших випадках - невизначено.

…. ….. ……

Π_n: $\delta_n(S, A_i) \quad = \quad \delta_{n-1}(S, A_i) \bigcup \{L \mid A_j \to \varpi_1 A_i \varpi_2, \quad L = First_k(w_2) \oplus_k L_p, L_p \in Local_k(S, A_j)\}$

в інших випадках - невизначено.

…. …. ….

Π_m: $\delta_m(S, A_i) = \delta_{m+1}(S, A_i) = ... \qquad \forall A_i \in N$

Тоді $Local_k(S, A_i) = \delta_m(S, A_i)$.

Виходячи з означення $Local_k(S, A_i)$, умови для *LL(k)*-граматики будуть наступними: для довільного *A*-правила виду:

$$A \to W_1 \mid W_2 \mid ... \mid W_p$$

$$First_k(W_i L_m) \quad \cap \quad First_k(W_j L_m) \quad = \quad O, \quad i \neq j \quad ma \quad L_m \in Local_k(S, A).$$

Як наслідок, з алгоритму пошуку $Local_k(S, A_i)$ видно, що

$$Follow_k(A_i) \quad = \quad \bigcup_{j=1}^{m} L_j, \quad L_j \in Local_k(S, A_i).$$

Для побудови синтаксичного аналізатора для *LL(k)*-граматики *(k>1)* необхідно побудувати множину таблиць, що забезпечать нам безступиковий аналіз вхідного ланцюжка *w* (програми) за час пропорційний *O(n)*, де $n = | w |$.

Побудову множини таблиць для управління *LL(k)*-аналізатором почнемо з таблиці, яка визначає перший крок безпосереднього виводу *w* в граматиці *G*:

$$T_0 = T_{s,\{\varepsilon\}}(u) = \begin{cases} -(T_1\alpha_1 T_2\alpha_2 ....T_p\alpha_p, n), \quad де\ n - номер\ правила\ виду\ S \to A_1\alpha_1 A_2\alpha_2 ....A_p\alpha_p, \\ \qquad A_i \in N, \quad i = 1,2,...p, \quad \alpha_i \in \Sigma^*, \quad i = 1,2,...p \\ \qquad u = First_k(A_1\alpha_1 A_2\alpha_2 ....A_p\alpha_p) \\ -\ в\ інших\ випадках - невизначено \end{cases}$$

Неформально, коли в магазині автомата знаходиться аксіома *S*, то нас цікавить перших *k* термінальних символів, які можна вивести з *S* (аксіома - поняття "програма") при умові, що після неї (програми) буде досягнуто *EOF*.

Імена інших таблиць $T_1, T_2, \ldots T_p$ визначаються так:

$$T_i = T_{A_i, L_i}, \quad де \quad L_i = First_k(\alpha_i A_{i+1}\alpha_{i+1} ....A_p\alpha_p), \quad i = 1,2,.., p$$

Наступні таблиці визначаються так:

$$T_i = T_{A_i,L_i}(u) = \begin{cases} -(T_1\alpha_1 T_2\alpha_2 ....T_p\alpha_p, n), \quad \text{де } n - \text{номер правила виду } A_i \to A_1\alpha_1 A_2\alpha_2 ....A_p\alpha_p, \\ \qquad A_i \in N, \quad i = 1,2,...p, \quad \alpha_i \in \Sigma^*, \quad i = 1,2,...p \\ \qquad u = First_k(A_1\alpha_1 A_2\alpha_2 ....A_p\alpha_p) \oplus_k L_i \\ - \text{в інших випадках} - \text{невизначено} \end{cases}$$

Наступні таблиці $T_1, T_2, ... T_p$ визначаються так:

$$T_j = T_{Aj,Lj}, \quad \text{де} \quad L_j = First_k(\alpha_j A_{j+1}\alpha_{j+1} ....A_p\alpha_p) \oplus_k L_i, \quad j = 1,2,...p$$

Виходячи з вищенаведеної побудови множини таблиць управління *LL(k)*-синтаксичним аналізатором видно, що для нетермінала A_i множина таблиць буде наступна:

$$T_p = T_{A_i,L_p}, \quad \text{де} \quad L_p \in Local_k(S, A_i)$$

Приклад. Побудувати множину таблиць управління для LL(2)-граматики з наступною схемою правил:

$S \to abA$ (1)

$S \to \varepsilon$ (2)

$A \to Saa$ (3)

$A \to b$ (4)

Для вищенаведеної граматики множини $First_2(A_i), \quad A_i \in N$ будуть такі:

$First_2(S) = \{ab, \varepsilon\}, \qquad First_2(A) = \{aa, ab, b\}$,

а множини $Local_2(A_i), \qquad A_i \in N : Local_2(S, S) = Local_2(S, A) = \{\{\varepsilon\}, \{aa\}\}$.

Побудуємо першу таблицю $T_0 = T_{S,\{\varepsilon\}}$. Для *S*-правила відповідні множини *u* будуть такі:

- $S \to abA$ (*правилро* 1)

  $u \in First_2(abA) = \{ab\}$
- $S \to \varepsilon$ (*правило* 2)

  $u \in First_2(\varepsilon) = \{\varepsilon\}$

Таблиця T_0 визначається так:

$T \setminus \Sigma^{*2}$	aa	ab	ba	bb	a	b	ε
$T_0 = T_{S,\{\varepsilon\}}$		$abT_1, 1$					$\varepsilon, 2$

Нова таблиця управління $T_1 = T_{A,\{\varepsilon\}}$. Для A-правила відповідні множини u будуть такі:

- $A \to Saa$ (*правило* 3)

  $u \in First_2(Saa) \oplus_2 \{\varepsilon\} = \{ab, aa\}$
- $A \to b$ (*правило* 4)

  $u \in First_2(b) \oplus_2 \{\varepsilon\} = \{b\}$

$T \setminus \Sigma^{*2}$	aa	ab	ba	bb	a	b	ε
$T_1 = T_{A,\{\varepsilon\}}$	$T_2aa, 3$	$T_2aa, 3$				$b, 4$	

Нова таблиця управління $T_2 = T_{S,L}, \quad \text{де} \quad L = First_2(aa) \oplus_2 \{\varepsilon\} = \{aa\}$. Для таблиці T_2 та *S*-правила множини *u* будуть такі:

– $S \to abA$ (*правилро* 1)

$u \in First_2(abA) = \{ab\} \oplus_2 \{aa\} = \{ab\}$

– $S \to \varepsilon$ (*правило* 2)

$u \in First_2(\varepsilon) = \{\varepsilon\} \oplus_2 \{aa\} = \{aa\}$

$T \setminus \Sigma^{*2}$	aa	ab	ba	bb	a	b	ε
$T_2 = T_{S,\{aa\}}$	ε, 2	abT_3, 1					

Наступна таблиця $T_3 = T_{A,L}$, *де* $L = First_2(\varepsilon) \oplus_2 \{aa\} = \{aa\}$. Для таблиці T_3 та A-правила множини *u* будуть такі:

– $A \to Saa$ (*правило* 3)

$u \in First_2(Saa) \oplus_2 \{aa\} = \{ab, aa\}$

– $A \to b$ (*правило* 4)

$u \in First_2(b) \oplus_2 \{aa\} = \{ba\}$

$T \setminus \Sigma^{*2}$	aa	ab	ba	bb	a	b	ε
$T_3 = T_{A,\{aa\}}$	T_2aa, 3	T_2aa, 3	b, 4				

Нова таблиця $T_4 = T_{S,L} = T_2$, *оскільки* $L = First_2(aa) \oplus_2 \{aa\} = \{aa\}$.

Ми визначили чотири таблиці-рядки (а їх кількість для довільної *LL(k)*-граматики визначається так:

$$\sum_{i=1}^{m} n_i, \quad де \quad n_i - кількість\ елементів\ множини\ Local_k(S, A_i), \quad m = |N|.$$

Об'єднаємо рядки-таблиці в єдину таблицю та виконаємо перейменування рядків:

$T \setminus \Sigma^{*2}$	aa	ab	ba	bb	a	b	ε
T_0		abT_1, 1					ε, 2
T_1	T_2aa, 3	T_2aa, 3				b, 4	
T_2	ε, 2	abT_3, 1					
T_3	T_2aa, 3	T_2aa, 3	b, 4				

Алгоритм. Синтаксичний аналізатор для *LL(k)*-граматики *(k>1)*.

$П_0$: Прочитати *k* лексем з вхідного файла програми (звичайно, інколи менше ніж *k*). В магазин занести таблицю T_0.

…. ….

$П_i$: - Якщо на вершині магазина знаходиться таблиця T_i, то елемент таблиці $M(T_i$, <k-вхідних лексем>) визначає ланцюжок, який T_i заміщає на вершині магазина.

- Якщо на вершині магазина $a_i \in \Sigma$ і перша поточна лексема з *k* прочитаних лексем рівна a_i, то з вершини магазина зняти a_i та прочитати з вхідного файла додатково одну лексему (звичайно, якщо це можливо).
- Якщо досягли кінця вхідного файла програми та магазин порожній, то програма не має синтаксичних помилок.

- В інших випадках - синтаксична помилка.

5. Лабораторний практикум побудови синтаксичних аналізаторів.

Граматика мови програмування визначається множиною БНФ-правил, що записані в текстовому файлі. Кожне провило обов'язково починається з першої

позиції рядка. \Для зручності правило можна продовжити в наступних рядках, але не з першої позиції. Нетермінал граматики - це ланцюжок літер, який починається з символу # та закінчується символом #, наприклад #програма#. Термінальні ланцюжки записуються традиційно, наприклад, begin. Альтернативи правила для зручності позначаються символом ! в першій позиції рядка, при цьому ліва частина правила опускається. Оскільки символи #, \ та ! є метасимволами при визначенні граматики, то для їх запису в термінальних ланцюжках використовується ще один метасимвол, а саме \. Правило граматики в текстовому файлі записується так:
#Pascal - програма# program (#список параметрів#) #блок# .

Лабораторний практикум другого семестру з дисципліни «Системне програмування» передбачає освоєння навиків створення власних програмних Java-проектів з використанням базового проекту, який повністю реалізований, та надається студенту у використання.

Визначимо основні положення базового проекту. Клас ***class MyLang*** забезпечує повне рішення та надає програмісту можливість замінити в екземплярі результуючого класу певне значення на власний результат та виконати проект в цілому. Якщо новостворена програмна система надає фінальний результат, який відповідає результату базового проекту, то алгоритм, який реалізований студентом, рахується вірним.

Таким чином, студенту надається «конструктор цеглин» якими можна скористатися на етапі реалізації власного алгоритму (звичайно, **не викликаючи «будівельний матеріал» базового проекту, який надає аналогічний результат**).

class MyLang

Тоді коротко, основні поля класу ***class MyLang*** наступні:

private int axioma;	// код аксіоми. В граматиці аксіома може бути не в першому правилі граматики
private boolean create;	// якщо екземпляп классу створено корректно, то true
private int LLK;	// значення LLK-контексту
private LinkedList<Node> language;	// список правил грамат
private LinkedList<TableNode> lexemaTable;	// таблиця перекодування лексем граматики
private int[] terminals;	// таблиця терміналів граматики
private int[] nonterminals;	// таблиця нетерміналів граматики
private int[] epsilonNerminals;	// таблиця ерсілон-нетерміналів граматики
private LlkContext[] termLanguarge;	// масив одно літерних словарних, побудованих на основі терміналів, тобто termLanguarge[i] – це одноелементна словарна множина, побудована на основі terminals[i]
private LlkContext[] firstK;	// массив словарних множин $first_k$, побудований выдповыдно до масиву nonterminals, тобто $first_k$[i] це множина $First_k$ для не термінала nonterminals[i].
private LlkContext[] followK;	// аналогічно попередньому поясненню
private LinkedList<LlkContext>[] LocalkContext;	// аналогічно попередньому поясненню, тобто LocalkContext[i] – це список множин $Local_k$ для нетермінала nonterminals[i].

private int[] uprTable;	// двомірний масив, який моделюється одноміпним масивом, у якого кількість стопців рівна (terminals.length+1), а рядків - nonterminals.length.

Конструктор екземплярів класу: public MyLang(String fileGramma, int llk).

Як бачимо, головний елемент (поле) класу ***MyLang*** LinkedList<Node> language – це список правил граматики. Кожне правило представляється екземпляром класу ***Node***. У загальному випадку, аксіома – не обов'язково лівий нетермінал першого правила. Вона визначається через метод public int getAxioma().

Множини $First_k$ (A_i), Followk(A_i), $First_k(w) \oplus_k Fllow_k(A_i)$ (для правила A_i -> w) – визначаються як екземпляри класу ***LlkContext***.

Множина Localk(S,A_i) визначається як ***LinkedList<LlkContext>***, тобто множина (список) елементів ***LlkContext.***

Для роботи з інформацією класу ***MyLang*** користувачу надається набір методів:

Група методів, яка забезпечує виконання алгоритмів:
public boolean isCreate() ; // Перевіряє, чи створено «complite» екземпляр класу ***MyLang***
public LinkedList<LlkContext>[] createLocalK() ;// побудова множин $Local_k(S,A_i)$
public boolean llkCondition(); // перевірка LL(k) - умови
public boolean createNonProdRools();// пошук непродуктивних правил
public boolean createNonDosNeterminals();// пошук недосяжних нетерміналів
public boolean leftRecursNonnerminal();// пошук ліво-рекурсивних нетерміналів
public boolean rightRecursNonnerminal();// пошук право-рекурсивних нетерміналів
public LlkContext[] firstK();(); // побудова множини $First_k$ (A_i),
public LlkContext[] followK();// побудова множини $Follow_k$ (A_i),
public void firstFollowK(); // побудова множини $First_k$ (w) $+_k$ $Follow_k$ (A_i) для всіх правил
public int[] createEpsilonNonterminals();// побудова множино e-нетерміналів
private int[] createTerminals(); // побудова множино терміналів
private int[] createNonterminals(); // побудова множино нетерміналів
public boolean strongLlkCondition() ; // перевірка сильноє LL(k)-умови
public int[] createUprTable() ; // побудова таблиці управління для LL(1)-граматики
public int[] createUprTableWithCollision(); // побудова таблиці управління для LL(1)-граматики за умови, що декілька не терміналів мають LL(1)- колізію, але граматика взагалі задовольняє сильній LL(2)-умові. Клітини таблиці управління, де є колізія позначаються (-1).

Група сервісних методів – забезпечує екстракцію даних з класу або запис даних до класу :
public int getAxioma();

public void setLocalkContext(LinkedList<LlkContext>[] localK) ; // дає можливість внести до екземпляр класу ***MyLang*** побудовану множину $Local_k$; Взагалі, в цьому проекті методи set забезпечують збереження результату в екземплярі класу ***MyLang***, а методи get – отримання результату.
public LinkedList<LlkContext>[] getLocalkContext();
public int[] getUprTable();
public LlkContext[] getFirstK();
public void setFirstK(LlkContext[] first) ;
public void setUprTable(int[] upr);
public LlkContext[] getFollowK();
public void setFollowK(LlkContext[] follow);
public int getLlkConst();
public LlkContext[] getLlkTrmContext(); // масив словарних множин для одноелементних термінвальних мов. І-та мова зберігає і-й термінал з масиву (множини) терміналів
public String getLexemaText(int code); //по коду лексеми ми визначаємо її текст
public int[] getTerminals()
public int[] getNonTerminals()
public int[] getEpsilonNonterminals()
public LinkedList<Node> getLanguarge()
public void setEpsilonNonterminals(int[] eps)

Група сервісних методів, які забезпечують друк (вивід у стандартний файл) результату:
public void printLocalk() ;
public void printTerminals();
public void printNonterminals();
public void prpintRoole(Node nod);
public void printFirstkContext()
public void printFirstFollowK()
public void printFirstFollowForRoole()
public void printFollowkContext()
public void printEpsilonNonterminals()
public void printGramma().

Class Node

Клас ***class Node*** забезпечує збереження та доступ до правил граматики у списку правил ***LinkedList<Node> language.*** Поля екземпляру класу ***Node:***
private int[] roole; // закодоване правило – це масив кодів лексем. Довжина масиву включає і лівий елемент правила. Нетермінали граматики закодовані від'ємними кодами, термінали – додатніми кодами.
private int teg; //робоче поле, доступне користувачу для маркування правил.
private LlkContext firstFollowK; // тут зберігається добуток $Firstk(w)+_k$ Followk(Ai) (для правила Ai-> w) для правила.

Для роботи з правилом граматики надаються наступні методи, їх семантика зрозуміла з попередніх визначень:
public void addFirstFollowK(LlkContext rezult);
public LlkContext getFirstFollowK() ;
public int[] getRoole();

Class LlkContext

Клас ***class LlkContext –*** відповідає за збереження словарних множин. Слово – це масив кодів лексем (int []). Е-слово має специфікацію new int[0] (мова Java допускає таку нотацію.

public LlkContext() ; // конструктор
public boolean wordInContext(int[] word) ; // слово є в словарній множині
public int[] getWord(int index) ; // отримати слово з індексом index з словарної множини. Індексація починається з нуля.
public int minLengthWord(); // мінімальна довжина слова у словарній множині
public int calcWords(); // кількість слів у словарній множині
public boolean addWord(int[] word) ; // добавляє слово до словарної множини. У випадку. Коли слово добавлено - значення методу **true**.

Class TableNode

Клас ***class TableNode –*** це допоміжний клас для зберігання текстів лексем граматики. Поля класу:
String lexemaText; // текст лексеми
int lexemacode; // код лексеми
static int numarator; // нумератор лексем

Маска коду не термінала 0x80000000, маска коду термінала 0x10000000.
В цьому класі важливий конструктор:
public TableNode (String lexema, int lexemaType);

Блок 1 Лабораторних робіт

1. Реалізувати препроцесор для вводу граматики в ЕОМ. Препроцесор перетворює послідовність правил текстового файлу в інформаційну структуру в пам'яті ЕОМ. Вхідні дані: файл з текстом граматики мови програмування. Результат занести до опису граматики - екземпляр класу *MyLang*;
2. Реалізувати алгоритм пошуку недосяжних не терміналів. Правила, що починаються з недосяжних не терміналів вилучити з граматики;
3. Реалізувати алгоритм пошуку непродуктивних правил. Непродуктивні правила вилучити з граматики.
4. Перевірити, чи L(G) – пуста множина.

5. Реалізувати алгоритм пошуку e-нетерміналів. Результат занести до опису граматики - екземпляр класу *MyLang*;
6. Реалізувати алгоритм пошуку ліво-рекурсивних не терміналів.
7. Реалізувати алгоритм пошуку право-рекурсивних не терміналів.
8. **Розробити та реалізувати алгоритм пошуку всіляких виводів мінімальної довжини (без повторів), які призводять до лівої рекурсії.
9. **Розробити та реалізувати алгоритм пошуку всіляких виводів мінімальної довжини (без повторів), які призводять до правої рекурсії.

Блок 2 Лабораторних робіт

1. Реалізувати алгоритм пошуку множин Firstk(Ai), Ai ∈ N. Результат занести до опису граматики - екземпляр класу *MyLang*.
2. Реалізувати алгоритм пошуку множин Follow(Ai), Ai ∈ N. Результат занести до опису граматики - екземпляр класу *MyLang*.
3. Реалізувати алгоритм пошуку множин Firstk(w) для правил Ai -> w, Ai ∈ N. Результат занести до опису граматики - екземпляр класу *MyLang*
4. Реалізувати алгоритм пошуку множин Firstk(w),$+_k$ Follow(Ai), для правил Ai -> w, Ai ∈ N. Результат занести до опису граматики - екземпляр класу *MyLang*.
5. Перевірити, чи є побудована граматика LL(1)-граматикою.
6. Перевірити, чи є побудована граматика сильною LL(k)-граматикою (для k=2);
7. Перевірити, чи є побудована граматика сильною LL(k)-граматикою (для k=3);
8. Побудувати таблицю управління LL(1)-синтаксичного аналізатора.
9. **Побудувати множини $Local_k$(S,Ai), Ai ∈ N. Результат занести до опису граматики - екземпляр класу *MyLang*.
10. **Перевірити, чи є граматика LL(k)-граматикою (k>1).

Блок 3 Лабораторних робіт

1. Реалізувати синтаксичний аналізатор мови Pl/0 методом магазинного автомата.
2. Реалізувати синтаксичний аналізатор мови Pl/0 методом рекурсивного спуску.
3. Реалізувати синтаксичний аналізатор мови Pl/0 методом рекурсивного спуску на правилах грамматики (рекурсивний алгоритм руху по правилах граматики);
4. Реалізувати синтаксичний аналізатор мови Pascal методом магазинного автомата.
5. Реалізувати синтаксичний аналізатор мови Pascal методом рекурсивного спуску.
6. Реалізувати синтаксичний аналізатор мови Pascal методом рекурсивного спуску на правилах грамматики (рекурсивний алгоритм руху по правилах граматики);
7. Реалізувати синтаксичний аналізатор мови C методом магазинного автомата (за умови наявності колізій для декількох не терміналів).
8. Реалізувати синтаксичний аналізатор мови C методом рекурсивного спуску (за умови наявності колізій для декількох не терміналів).
9. Реалізувати синтаксичний аналізатор мови C методом рекурсивного спуску на правилах грамматики (рекурсивний алгоритм руху по правилах граматики) за умови наявності колізій для декількох не терміналів;

Блок 4 Лабораторних робіт

Наведений нижче перелік лабораторних робіт - це дослідницькі роботи, які передбачають вивчення додаткового матеріалу та практичних навиків попередніх розділів.

1. Побудувати LL(1)-граматику для мови програмування Java. Реалізувати синтаксичний аналізатор мови програмування Java.
2. Побудувати LL(1)-граматику для select – statement мови програмування SQL. Реалізувати синтаксичний аналізатор мови select – statements.
3. Побудувати LL(1)-граматику для delete – statement мови програмування SQL. Реалізувати синтаксичний аналізатор мови delete – statements.
4. Побудувати LL(1)-граматику для insert – statement мови програмування SQL. Реалізувати синтаксичний аналізатор мови insert – statements.
5. Побудувати LL(1)-граматику для update – statement мови програмування SQL.
   Реалізувати синтаксичний аналізатор мови update – statements.
6. Реалізуйте препроцесор мови програмування C++.
7. Реалізуйте препроцесор мови програмування C.
8. Скористатися системою JCC (Java Compiler Compiler). Випишіть граматику Pascal для JCC.

Література.

1. Агафонов В.Н. Синтаксический анализ языков программирования. Новосибирск. Из-во НГУ. 1981.
2. Ахо А. Ульман Дж. Теория синтаксического анализа, перевода и компиляции. Т1. М. Мир. 1978.
3. Братчиков И.А. Синтаксис языков программирования. М. Наука. 1975.
4. Вайнгартен Ф. Трансляция языков программирования. М. Мир. 1977.
5. Вирт Н. Систематическое программирование. Введение. М.Мир. 1977.
6. Глушков В.М., ЦейтлинГ.Е., Ющенко Е.Л. Алгебра, языки, программи-рование. Киев. Наукова думка. 1974.
7. Ингерман П. Синтаксически ориентированный транслятор. М. Мир. 1969.
8. Лебедев В.Н. Введение в системы программирования. М. Статистика. 1975.
9. Миккиман У., Хорнинг Дж., Уортман Д. Генератор компиляторов. М. Статистика. 1980.
10. Пратт Т. Языки программирования: разработка и реализация. М. Мир. 1979.
11. Чантер Р. Проектирование и конструирование компиляторов. М. финансы и статистика. 1984.
12. Грис Д. Построение компиляторов для ЦЭВМ. М. Мир. 1976.
13. Бек Д. Введение в системное программирование. М. Мир. 1988.

14. Льюис Ф., Стирнз Р., Розенкранц Д. Теоретические основы построения компиляторов. М. Мир. 1979.