

Зміст

1. Мови програмування та мовні процесори	2
2. Лексичний аналіз в мовних процесорах	4
2.1 Скінчені автомати	5
2.2 Мінімізація детермінованих скінчених автоматів.....	7
2.3 Скінчені автомати та праволінійні граматики.....	10
2.4 Регулярні множини та регулярні вирази	12
2.5 Польський інверсний запис для регулярних виразів	14
2.6 Інтерпретація ПОЛІЗ регулярного виразу.	15
2.7 Застосування скінчених автоматів при розробці лексичних аналізаторів	15
2.8 Методика програмування лексичних аналізаторів на основі скінчених автоматів.....	20
3. Лабораторний практикум побудови лексичних аналізаторів.	22
4. Синтаксичний аналіз в мовних процесорах	23
3.1. Магазинні автомати.....	26
3.2. Синтаксичний аналіз без повернення назад.....	27
3.3. Синтаксичний аналіз на основі $LL(1)$-граматик.....	34
3.4. $LL(1)$-синтаксичний аналізатор для мови Pascal.....	36
3.5. Метод рекурсивного спуску програмування синтаксичних аналізаторів	38
3.6. Побудова $LL(k)$-синтаксичного аналізатора ($k>1$).	42
5. Лабораторний практикум побудови синтаксичних аналізаторів.....	45
5.1. Блок №1 лабораторних робіт	47
5.2. Блок №2 лабораторних робіт	49
5.3. Блок №3 лабораторних робіт	52
Література.	52

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 семантики. 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.

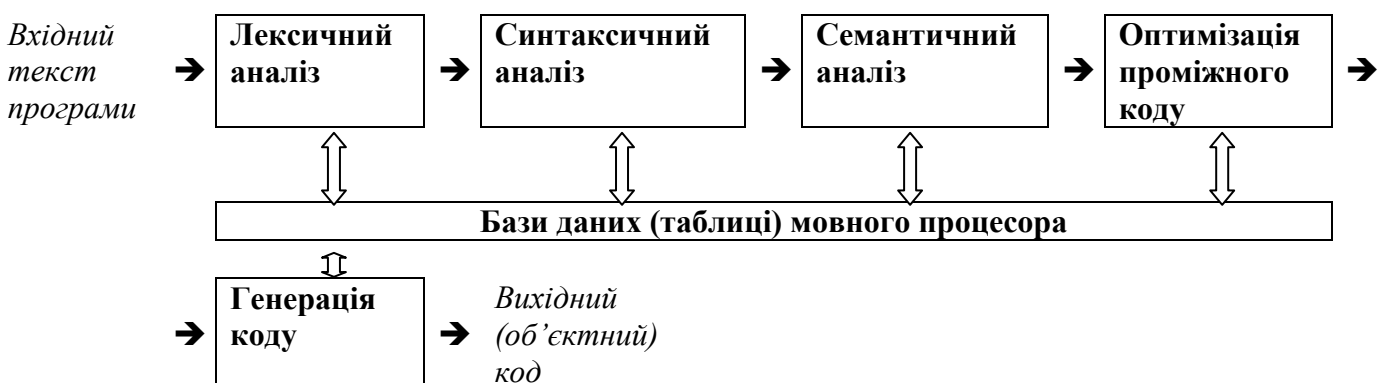
Мовний процесор типу транслятор (транслятор) – це програмний комплекс, котрий на вході отримує текст програми на вхідній мові, а на виході видає версію програми на вихідній мові, що називається об'єктною мовою. В більшості випадків як об'єктна мова виступає мова команд деякої обчислювальної машини. Серед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

$$M = \langle Q, \Sigma, \delta, q_0, F \rangle, \text{ де}$$

- $Q = \{q_0, q_1, \dots, q_{n-1}\}$ – скінчена множина станів автомата;
- $\Sigma = \{a_1, a_2, \dots, a_m\}$ – 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 заклучними 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:

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

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

$$(q_0, w) \mid=^* (q, \epsilon) \text{ для деякого } q \in F, \text{ де}$$

$\mid=^*$ - рефлексивно-транзитивне замикання бінарного відношення $\mid=$.

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

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

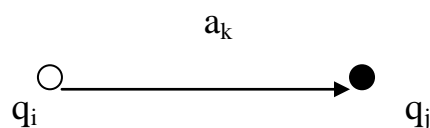
На практиці, при визначенні скінченого автомата M , використовують декілька способів визначення функції δ , наприклад:

- 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

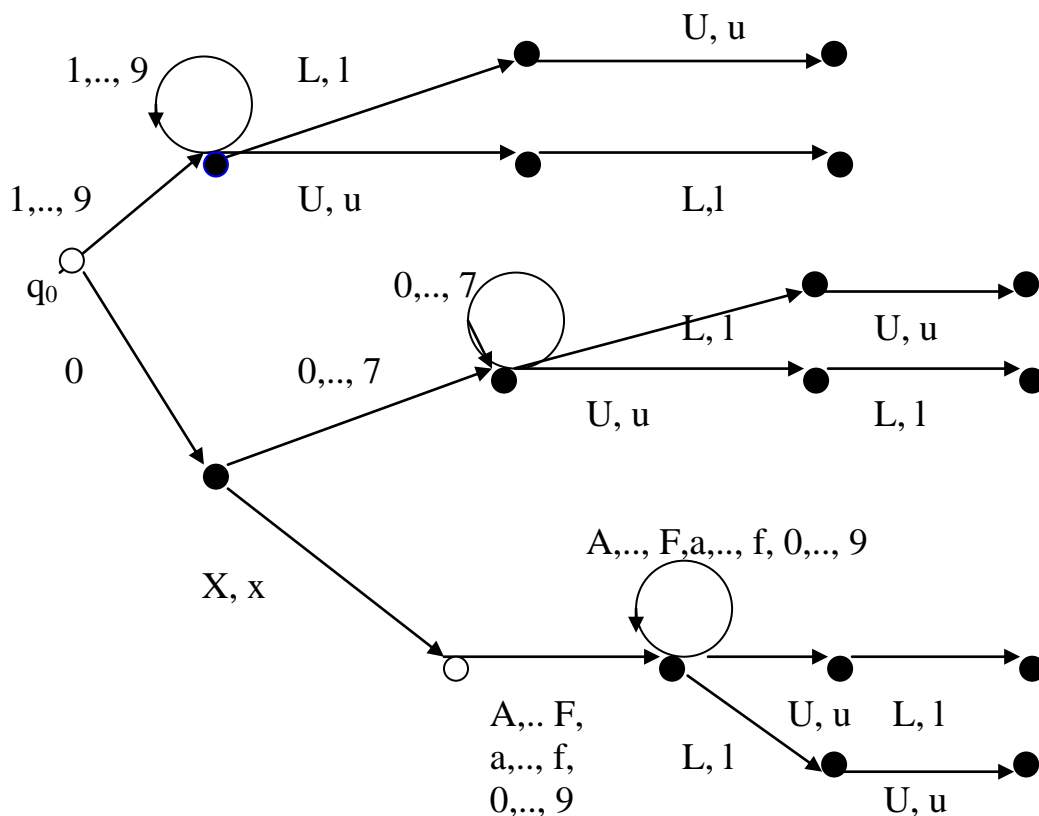
$$M(q_i, a_j) = \{ q_k \mid q_k \in \delta(q_i, a_j) \}$$

Діаграма переходів скінченого автомата M - це неупорядкований 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: $\bullet q_i$.

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



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 скінченного 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

$$M = \langle Q, \Sigma, \delta, q_0, F \rangle$$

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

1. $Q_1 = P(Q)$, тобто імена станів автомата M_1 – 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 $S \cap A \neq \emptyset$.
4. $\delta_1(S, a) = \{ q \mid q \in \delta(q_i, a), q_i \in S \}$.

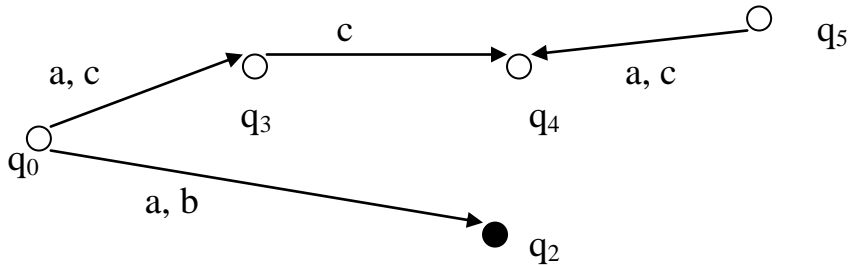
Доведемо індукцією по i , що $(S, w) \stackrel{i}{=} (S_1, \varepsilon)$, тоді і тільки тоді, коли $S_1 = \{ q \mid (q_i, w) \stackrel{i}{=} (q, \varepsilon), \text{ для } q_i \in S \}$,

Зокрема, $(\{q_0\}, w) \stackrel{*}{=} (S_1, \varepsilon)$, для деякого $S_1 \in F_1$, тоді і тільки тоді, коли $(q_0, w) \stackrel{*}{=} (q, \varepsilon)$, $q \in F$. 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 $q_5 \in$ "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 $Q \setminus Q_m$ - множина недосяжних станів. Побудуємо послідовність множин Q_0, Q_1, Q_2, \dots таким чином, що:

П0. $Q_0 = \{ q_0 \}$.

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

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

.....
 Пm. $Q_m = Q_{m+1} = \dots$

Очевидно, що кількість кроків П0, П1... скінчена, тому що послідовність Q_i монотонна ($Q_0 \subseteq Q_1 \subseteq Q_2 \subseteq \dots$) та обмежена зверху - $|Q_m| \leq |Q|$.

Тоді Q_m - множина досяжних станів скінченого автомата, а $Q \setminus Q_m$ - 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 $Q \setminus S_m$ - множина тупикових станів. Побудуємо послідовність множин S_0, S_1, S_2, \dots таким чином, що:

П0. $S_0 = \{ q \mid q \in F \}$.

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

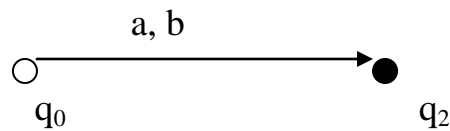
П2. $S_i = S_{i-1} \cup \{ q \mid \delta(q, a) \cap S_{i-1} \neq \emptyset, a \in \Sigma \}$.

.....
 Пм. $S_m = S_{m+1} = \dots$

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

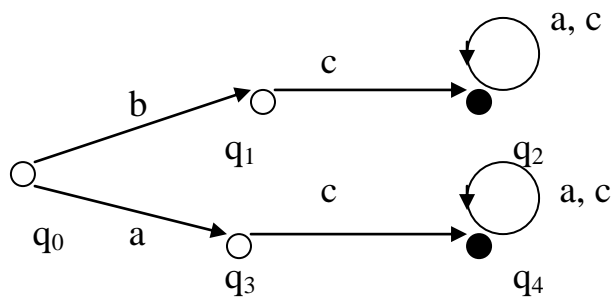
Тоді S_m – множина нетупикових станів скінченого автомата, а $Q \setminus S_m$ – множина тупикових станів. В новому (мінімізованому) автоматі функція δ визначається лише для нетупикових станів.

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:



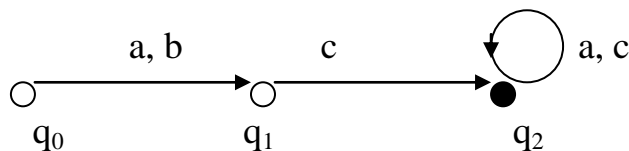
Мал. 3.

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 $(q_1, x) \models^* (q_3, \epsilon)$ та $(q_2, x) \not\models^* (q_3, \epsilon)$, 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 ($|x| \leq k$), що розрізняє стани q_1 та q_2 . Два стани q_1 та q_2 нерозрізнені, якщо вони k -нерозрізнені для довільного k .

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

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

Побудуємо відношення \equiv^k :

$q_1 \equiv^k q_2$, якщо $q_1 \equiv^{k-1} q_2$ та $\delta(q_1, a) \equiv^{k-1} \delta(q_2, a)$ 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 скінченного 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 уявити, які мови (словарні множини) є скінченоавтоматними.

Твердження: Скінчено автоматними є наступні множини:

- порожня словарна множина - \emptyset ;
- словарна множина, що складається з одного ϵ -слова – $\{\epsilon\}$;
- 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 \emptyset ;

2. Розглянемо автомат $M = \langle \{q_0\}, \Sigma, q_0, \delta, \{q_0\} \rangle$, у якому δ не визначено ні для яких $a \in \Sigma$.

Тоді $L(M) = \{\epsilon\}$.

3. Розглянемо автомат $M = \langle \{q_0, q_1\}, \Sigma, q_0, \delta, \{q_1\} \rangle$, 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 $M_1 = \langle Q_1, \Sigma, q_{01}, \delta_1, F_1 \rangle$ та $M_2 = \langle Q_2, \Sigma, q_{02}, \delta_2, F_2 \rangle$, що визначають відповідно мови $L(M_1)$ та $L(M_2)$, то скінченоавтоматними мовами будуть:

- $L(M_1) \cup L(M_2)$;
- $L(M_1) * L(M_2)$;
- $\{L(M_1)\} = \{\epsilon\} \cup L(M_1) \cup L(M_1) * L(M_1) \cup \dots$,

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 $M = \langle Q, \Sigma, q_0, \delta, F \rangle$, 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:
 $\delta(q,a) = \delta_1(q,a)$, $q \in Q_1$, $a \in \Sigma$;
 $\delta(q,a) = \delta_2(q,a)$, $q \in Q_2$, $a \in \Sigma$;
 $\delta(q_0,a) = \delta_1(q_{01},a) \cup \delta_2(q_{02},a)$, $q_{01} \in Q_1$, $q_{02} \in Q_2$, $a \in \Sigma$;
- Множина заключних станів $F = \left\{ \begin{array}{l} F_1 \cup F_2, \text{ якщо } \varepsilon \notin (L \cup L_2) \\ F_1 \cup F_2 \cup \{q_0\}, \text{ якщо } \varepsilon \in (L \cup L_2) \end{array} \right\}$

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

2. Побудуємо автомат $M = \langle Q, \Sigma, q_0, \delta, F \rangle$, 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:
 $\delta(q,a) = \delta_1(q,a)$, $q \in Q_1 \setminus F_1$, $a \in \Sigma$;
 $\delta(q,a) = \delta_2(q,a)$, $q \in Q_2$, $a \in \Sigma$;
 $\delta(q,a) = \delta_1(q,a) \cup \delta_2(q_{02},a)$, $q \in F_1$, $q_{02} \in Q_2$, $a \in \Sigma$;
- Множина заключних станів $F = \left\{ \begin{array}{l} F_2, \text{ якщо } \varepsilon \notin L_2 \\ F_1 \cup F_2, \text{ якщо } \varepsilon \in L_2 \end{array} \right\}$

3. Побудуємо автомат $M = \langle Q, \Sigma, q_0, \delta, F \rangle$, такий що $L(M) = \{L(M_1)\}$:

- $Q = Q_1 \cup \{q_0\}$, де q_0 – новий стан ($q_0 \notin Q_1$);
- Функцію δ визначимо таким чином:
 $\delta(q,a) = \delta_1(q,a)$, $q \in Q_1 \setminus F_1$, $a \in \Sigma$;
 $\delta(q_0,a) = \delta_1(q_{01},a)$, $q_{01} \in Q_1$, $a \in \Sigma$;
 $\delta(q,a) = \delta_1(q,a) \cup \delta_1(q_{01},a)$, $q \in F_1$, $a \in \Sigma$;
- Множина заключних станів $F = F_1 \cup \{q_0\}$.

2.3 Скінчені автомати та праволінійні граматики

Означення: Породжуюча граMATика G – це п'ятірка

$$G = \langle N, \Sigma, P, S \rangle,$$

де: N – скінчена множина - допоміжний алфавіт (нетерміналі);

Σ – скінчена множина – основний алфавіт (терміналі);

P – скінчена множина правил виду

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

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

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

- Тип 1: граматики, що не укорочуються, коли обмеження на правила мінімальні, а саме:

$$\alpha \rightarrow \beta, \alpha \in (N \cup \Sigma)^* * N * (N \cup \Sigma)^*, \beta \in (N \cup \Sigma)^*, |\alpha| \leq |\beta|.$$

- Тип 2: контекстно-вільні граматики, коли правила в схемі P мають вигляд:

$$A_i \rightarrow \beta, A_i \in N, \beta \in (N \cup \Sigma)^*.$$

- Тип 3: скінченоавтоматні граматики, коли правила в схемі P мають вигляд:

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

$$A_i \rightarrow w, w \in \Sigma^*;$$

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

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

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

$$A_i \rightarrow w, w \in \Sigma^*;$$

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 $w = \alpha\alpha$, $w_1 = \alpha\beta$ та в схемі P граматики G є правило виду $\alpha \rightarrow \beta$. 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 $w \Rightarrow w^1, w^1 \Rightarrow w^2, \dots, w^{n-1} \Rightarrow w_1$. Або кажуть, що бінарне відношення \Rightarrow^* - це рефлексивно-транзитивне замикання бінарного відношення \Rightarrow .

3. Мова, яку породжує граMATика 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:

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

$$- A_i \rightarrow w, w \in \Sigma^*; \quad (2)$$

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

- правила виду $A_i \rightarrow a_1a_2\dots a_pA_j$ замінимо послідовністю правил:

$$A_i \rightarrow a_1B_1$$

$$B_1 \rightarrow a_2B_2$$

$$\dots\dots\dots$$

$$B_{p-1} \rightarrow a_pA_j ;$$

- правила виду $A_i \rightarrow a_1a_2\dots a_p$ замінимо послідовністю правил:

$$A_i \rightarrow a_1B_1$$

$$B_1 \rightarrow a_2B_2$$

$$\dots\dots\dots$$

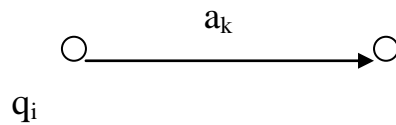
$$B_{p-1} \rightarrow a_p B_p$$

$$B_p \rightarrow \varepsilon$$

де: B_1, B_2, \dots - це нові нетермінальні символи граматики G_1 . Очевидно, що граMATика G_1 буде еквівалентна граматиці G .

Далі, на основі граматики G_1 побудуємо скінчений автомат M , таким чином:

- як імена станів автомата візьмемо нетермінальні символи граматики G_1 ;
- 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 $(q_0, w) \models^n (q, \varepsilon)$:

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

Нехай $|w| = i+1$, тобто $w = a w_1$: тоді $S \Rightarrow a A_p \Rightarrow^i a w_1$ та $(q_0, a w_1) \models (q_i, w_1) \models^{i-1} (q, \varepsilon)$, $q \in F$.

Доведення навпаки є очевидним.

2.4 Регулярні множини та регулярні вирази

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

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

П2. $\{\varepsilon\}$ – 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:

$P \cup Q$ (операція об'єднання);

$P * Q$ (операція конкатенації);

$\{P\} = \{\varepsilon\} \cup P \cup P * P \cup \dots$ (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. \emptyset позначає регулярну множину \emptyset ;

П2. ε позначає регулярну множину $\{\varepsilon\}$;

П3. a позначає регулярну множину $\{a\}$;

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

$p + q$ позначає регулярну множину $P \cup Q$;

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

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

$$X = aX + b, \text{ де } a, b - \text{регулярні вирази.}$$

Взагалі таке рівняння в залежності від a та b може мати безліч розв'язків, наприклад: $X = a^*(b + \gamma)$, при умові, що a 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 найменша нерухома точка. 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:

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

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

$$\begin{aligned} X_1 &= a_{11}X_1 + a_{12}X_2 + \dots + a_{1n}X_n + b_1 \\ X_2 &= a_{21}X_1 + a_{22}X_2 + \dots + a_{2n}X_n + b_2 \\ &\dots \\ X_n &= a_{n1}X_1 + a_{n2}X_2 + \dots + a_{nn}X_n + b_n \end{aligned}$$

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:

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

П3. Збільшити i 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 правій частині рівняння для X_i не буде невідомих X_1, X_2, \dots, X_{i-1} . Перейти на П5, при цьому i 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, \dots, X_{i-1} підставити замість X_i регулярний вираз $a*b$.

П6. Якщо i рівне 1 то зупинитися, інакше i 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 виразі 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 (\cdot) 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 ПОЛІЗ – це скінчений автомат M , який розпізнає (сприймає) множину ланцюжків, котрі позначає регулярний вираз.

П0. Причитати поточну лексему.

П1. Якщо поточна лексема операнд a_i , то побудувати скінчений автомат M , 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 M 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 автомат, 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 S можна визначити як скінчену множину

$$L_0 = \{ +, -, /, *, \dots \}$$

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

Програму, яка розпізнає множину лексем, реалізуємо двома способами:

- перший спосіб: створимо програмний модуль, роботою котрого керує таблиця управління скінченого автомата $M(q_i, a_j)$, яка визначається в програмі явно;
- другий спосіб: 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[ ]="0123456789ABCDEFabcdefXxLIUu"; // 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, 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, 4, 4}, // q2=3
```



```

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 C. 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 – невiрне продовження шiстнадцяткової 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 ASCII: 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 ASCII: `\`, `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[]=

```

```

        'ABCDEFGHIJKLMNOPQRSTUVWXYZ_abcdefghijklmnopqrstuvxyz'
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 = fgetc(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

```

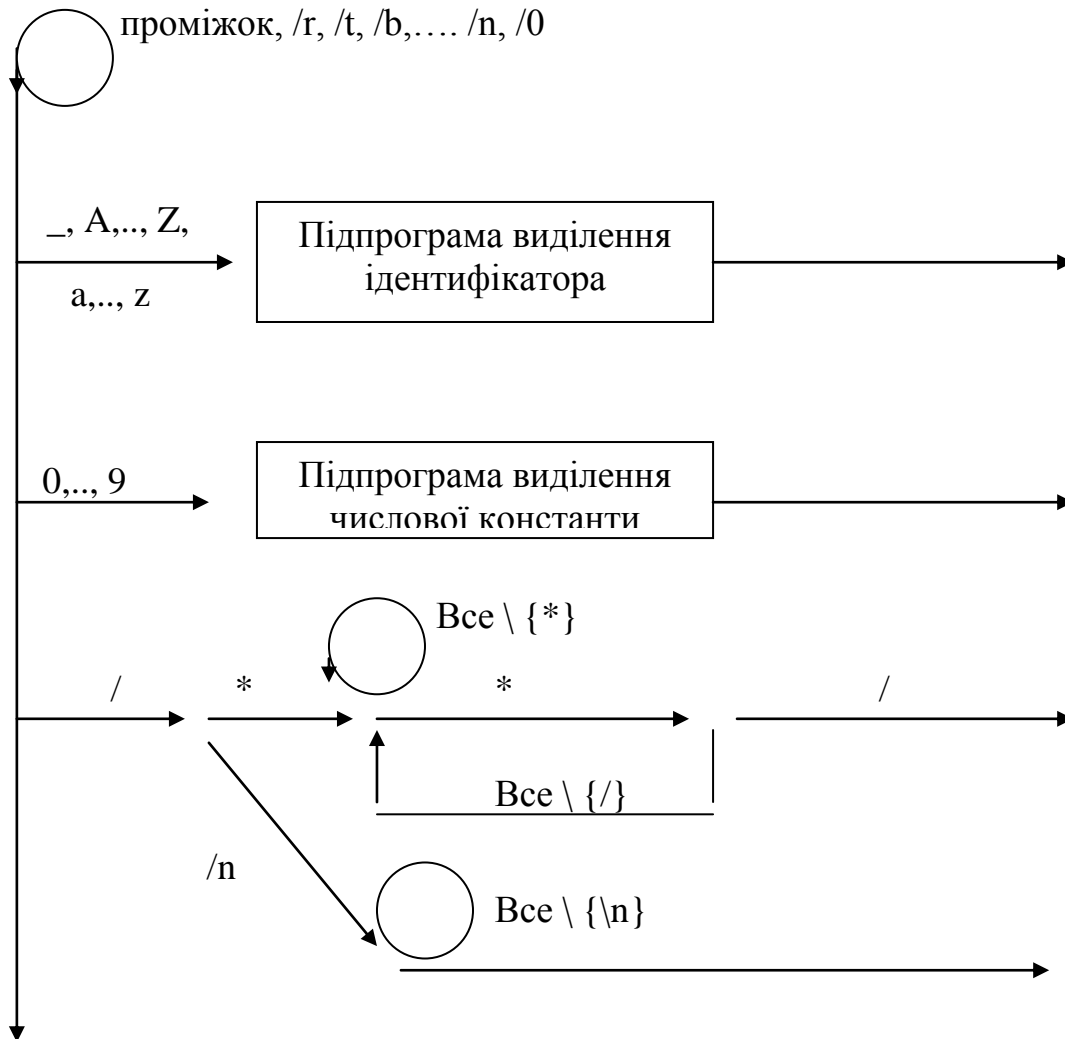
2.8 Методика програмування лексичних аналізаторів на основі скінчених автоматів.

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 скінченного 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 .

$$S \rightarrow S + S \mid S * S \mid a$$

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 .

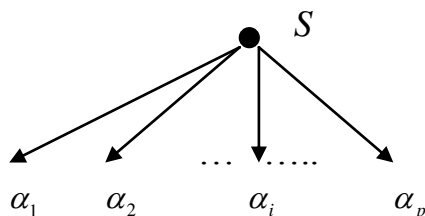
Алгоритм. Побудова синтаксичного дерева ланцюжка ω в граматиці G з урахуванням лівосторонньої стратегії виводу:

П₁. Будуємо корінь дерева та позначимо його аксіомою S .

П₂. В схемі P граматики G візьмемо правило виду $S \rightarrow \alpha_1 \alpha_2 \dots \alpha_p$, 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 $\alpha_i \rightarrow \beta_1 \beta_2 \dots \beta_l$, де $\beta_i \in N \cup \Sigma \cup \{\varepsilon\}$ та побудуємо наступне дерево (Мал. 2).

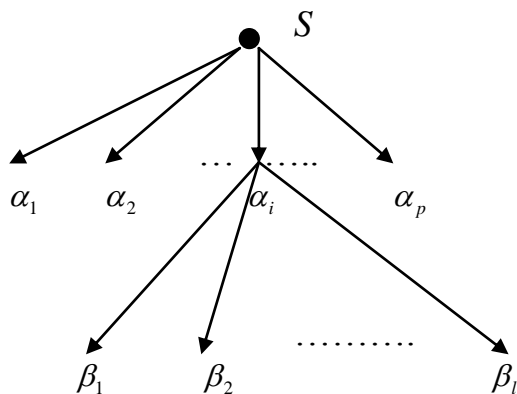


Мал. 1.

Пункт Π_i виконувати доти, доки на кроні дерева будуть елементи з N .

Відмітимо очевидні факти, що впливають з побудови синтаксичного дерева:

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



Мал. 2.

Означення. Будемо говорити, що ланцюжок $\omega \in \Sigma^*$, побудований на основі граматки $G(\omega \in L(G))$ 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 граматки $G = \langle N, \Sigma, P, S \rangle$ зі схемою P

$$S \rightarrow S + T \quad (1)$$

$$S \rightarrow T \quad (2)$$

$$T \rightarrow T * F \quad (3)$$

$$T \rightarrow F \quad (4)$$

$$F \rightarrow (S) \quad (5)$$

$$F \rightarrow a \quad (6)$$

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

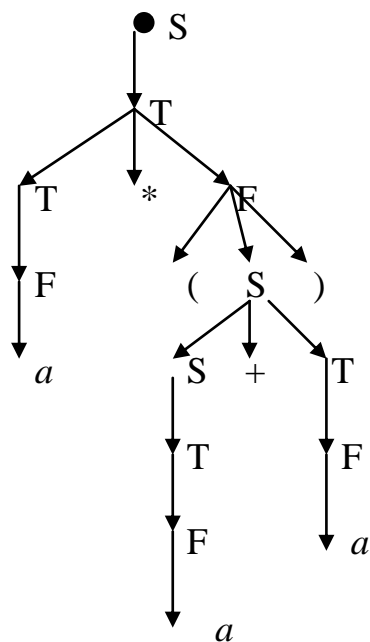
$$S \Rightarrow T \Rightarrow T * F \Rightarrow F * F \Rightarrow a * F \Rightarrow a * (S) \Rightarrow a * (S + T) \Rightarrow a * (T + T) \Rightarrow$$

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

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".



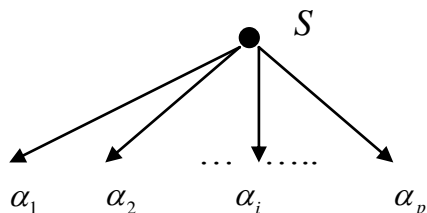
Мал. 4

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 $\omega \in L(G)$.

Π_0 : Побудуємо корінь дерева та позначимо його аксіомою S . Тоді, якщо $\pi = \langle p_1, p_2, \dots, p_m \rangle$, то

Π_1 : Побудуємо дерево висоти один, взявши зі схеми P правило з номером p_1 виду $S \rightarrow \alpha_1 \alpha_2 \dots \alpha_p$ (Мал. 5):



Мал. 5.

Π_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: $\alpha_i \rightarrow \beta_1 \beta_2 \dots \beta_l$ 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 граMATИКА, у якій в схемі P існує така послідовність правил за участю нетермінала A_i , що:

$$A_i \rightarrow A_j, \text{ де } A_i \text{ — будь-який нетермінал граматики } G.$$

$$A_j \rightarrow A_i$$

- як наслідок з попереднього пункту, граматики з ліворекурсивним нетерміналом для стратегії аналізу "зверху донизу" недопустимі.

- існують підкласи класу КС-грамматик, які природно забезпечують стратегію аналізу "зверху донизу". Один з таких підкласів — це LL(k)-грамматики, 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: $Q = \{q_0, q_1, \dots, q_{m-1}\}$ — множина станів магазинного автомату;

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

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

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

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

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

$F \subseteq Q$ — 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:

$(q_1, a\omega, jr) \models (q_2, \omega, r_1r)$ 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 $(q_1, \omega_1, j_1) \models (q_1^1, \omega_1^1, j_1^1)$, $(q_1^1, \omega_1^1, j_1^1) \models (q_1^2, \omega_1^2, j_1^2), \dots, (q_1^{n-1}, \omega_1^{n-1}, j_1^{n-1}) \models (q_2, \omega_2, j_2)$.

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 мови $L(M)$ 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 магазину.
- функцію σ визначимо так:

- якщо $A \rightarrow \omega_1 | \omega_2 | \dots | \omega_p$ належить P , то

$$\sigma(q_0, \varepsilon, A) = \{(q_0, \omega_1), (q_0, \omega_2), \dots, (q_0, \omega_p)\}.$$

- 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 $S \Rightarrow x_1 x_2 \dots x_k$ тоді МП-автомат з початкової конфігурації (q_0, ω, S) перейде в наступну конфігурацію $(q_0, \omega, x_1 x_2 \dots x_k)$. Далі розглянемо наступні ситуації:

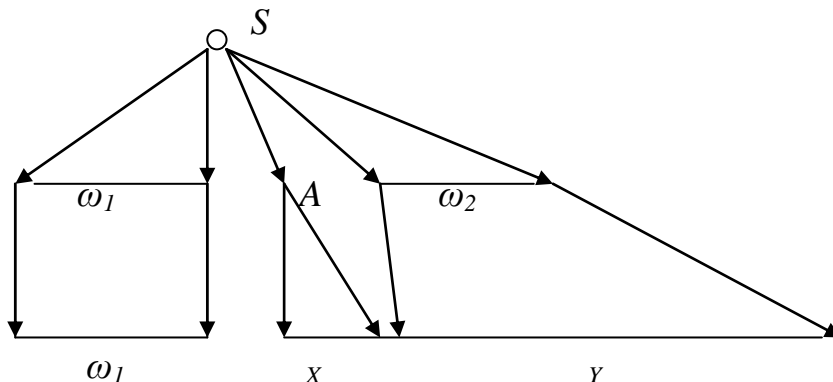
- коли x_1 — термінал a_1 (тобто $\omega = a_1 \omega_1$), тоді МП-автомат виконає наступний такт: $(q_0, a_1 \omega_1, a_1 x_2 \dots x_k) \models (q_0, \omega_1, x_2 \dots x_k)$;
- коли x_1 — нетермінал, тоді в схемі P грамматики G виберемо правило виду $x_1 \rightarrow y_1 y_2 \dots y_l$, зробимо наступний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

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

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.

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

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 $LL(k)$ -грамматик — це підклас КС-грамматик, який не покриває його.

Продемонструємо на прикладі справедливості твердження 4. Розглянемо грамматику G з наступною схемою P :

$$S \rightarrow Sa | b$$

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

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

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

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

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

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

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

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):

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

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

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

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

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

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

П₁. $F_i(a) = \{a\}$ для всіх $a \in \Sigma$, $i \geq 0$.

П₂. $F_0(A_i) = \{\omega | \omega \in \Sigma^{*k} \text{ для } A_i \rightarrow \omega x \text{ при } (|\omega| = k \wedge x \neq \varepsilon) \vee (|\omega| \leq k \wedge x = \varepsilon)\}$

в інших випадках - невизначено.

П_n.

$$F_n(A_i) = F_{n-1}(A_i) \cup \{\omega | \omega \in \Sigma^{*k}, \omega \in F_{n-1}(\alpha_1) \oplus_k \dots \oplus_k F_{n-1}(\alpha_p)\}$$

для правила $A_i \rightarrow \alpha_1\dots\alpha_p$, за умови, що $F_{n-1}(\alpha_1), F_{n-1}(\alpha_2), \dots, F_{n-1}(\alpha_p)$ визначені

в інших випадках - невизначено.

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

Очевидно, що:

- послідовність $F_0(A_i) \subseteq F_1(A_i) \subseteq \dots$ — 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 граматки з наступною схемою правил:

$S \rightarrow BA$
 $A \rightarrow +BA \mid \varepsilon$
 $B \rightarrow DC$
 $C \rightarrow *DC \mid \varepsilon$
 $D \rightarrow (S) \mid a$

Нехай $k=2$.

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

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

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

$$First_k(\alpha \cdot \omega_2) \cap First_k(\beta \cdot \omega_2) = \emptyset \quad (1)$$

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

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

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

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 $A \rightarrow \alpha \mid \beta$ задовольняється умова:

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

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

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

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

Без доведення зафіксуємо наступні твердження:

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

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

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

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 $S \rightarrow aAaa \mid bAba$

$$\begin{aligned} & First_2(aAaa \cdot Follow_2(S)) \cap First_2(bAba \cdot Follow_2(S)) = \\ & = (First_2(aAaa) \oplus_2 Follow_2(S)) \cap (First_2(bAba) \oplus_2 Follow_2(S)) = \\ & = (\{ab, aa\} \oplus_2 \{\varepsilon\}) \cap (\{bb\} \oplus_2 \{\varepsilon\}) = \{ab, aa\} \cap \{bb\} = \emptyset \end{aligned}$$

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

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

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

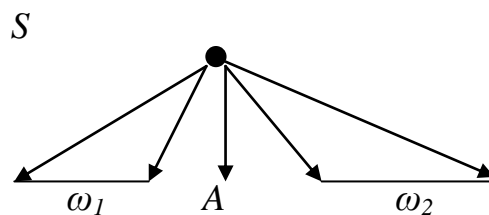
а) $S \Rightarrow^* aAaa$
 $First_2(b \cdot aa) \cap First_2(\varepsilon \cdot aa) = \{ba\} \cap \{aa\} = \emptyset$

б) $S \Rightarrow^* bAba$
 $First_2(b \cdot ba) \cap First_2(\varepsilon \cdot ba) = \{bb\} \cap \{ba\} = \emptyset$

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

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

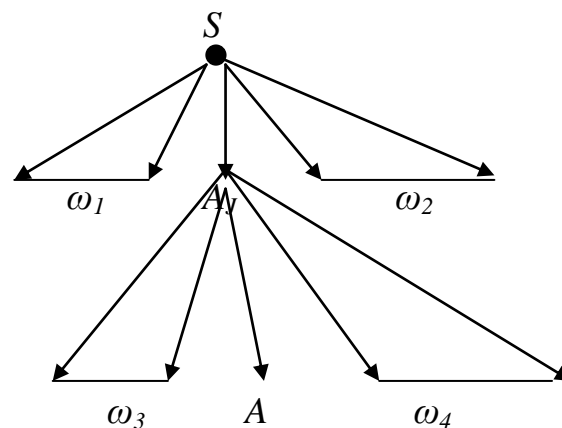
Для побудови алгоритму пошуку множини $Follow_k(A_i)$, $A_i \in N$ розглянемо наступні приклади на синтаксичному дереві, які дозволять перейти до узагальнень. Подивимося на синтаксичне дерево висоти 1 для правила $S \rightarrow \omega_1 A \omega_2$



Мал. 6

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

Далі розглянемо дерево висоти 2:



Мал. 7.

Тоді $First_k(\omega_1\omega_2) \subseteq Follow_k(A)$. В силу вищесказаного, будемо знаходити значення функції $\sigma(S, A_i)$, тобто будемо розглядати всілякі дерева, які можна побудувати, починаючи з аксіоми S .

Π_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 $A_i \in N \setminus \{S\}$.

Π_1 . $\sigma_1(S, A_i) = \sigma_0(S, A_i) \cup \{\omega | S \rightarrow \omega_1 A_i \omega_2, \omega \in First_k(\omega_2)\}$. В інших випадках $\sigma_1(S, A_i)$ — невизначено.

....

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

....

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

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

Очевидно, що:

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

До того, як перевірити граматiku 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:

$$S_0 = \{A_i | A_i \rightarrow \varepsilon\}$$

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

....

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

.... Π_n

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

Тоді множина S_m — множина ε -нетерміналів.

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

$$S \rightarrow a B D \quad | \quad D \quad | A C \quad | b$$

$$A \rightarrow S C B \quad | S A B C \quad | C b D \quad | \varepsilon$$

$$B \rightarrow C A \quad | d$$

$$C \rightarrow A D C \quad | a \quad | \varepsilon$$

$$D \rightarrow E a C \quad | S C$$

$$E \rightarrow B C S \quad | a$$

$$S_0 = \{A, C\}$$

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

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

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

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

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

Алгоритм. Тестування нетерміналу A_i на ліву рекурсію. Для кожного нетерміналу A_i побудуємо наступну послідовність множин S_0, S_1, \dots .

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

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

....

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

....

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

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

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

$S \rightarrow A b S \mid A C$

$A \rightarrow B D$

$C \rightarrow S a \mid \varepsilon$

$B \rightarrow B C \mid \varepsilon$

$D \rightarrow a B \mid B A$

1. Виконаємо процедуру тестування для кожного нетерміналу окремо:

- наприклад, для нетерміналу 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 граматичку зі схемою правил $S \rightarrow Sa \mid b$, яка має ліворекурсивний нетермінал S . Замінімо схему правил новою схемою з трьома правилами

$S \rightarrow bS_1$

$S_1 \rightarrow aS_1 \mid \varepsilon$.

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

$S \rightarrow BA$

$A \rightarrow +BA \mid \varepsilon$

$B \rightarrow DC$

$C \rightarrow *DC \mid \varepsilon$

$D \rightarrow (S) \mid a$

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

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

$\delta_n A_i$	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	{ $\epsilon,)$ }	{ ϵ }	{+, ϵ }	{+, ϵ }	{*, +, ϵ }
δ_5	{ $\epsilon,)$ }	{ $\epsilon,)$ }	{+, ϵ }	{+, $\epsilon,)$ }	{*, +, ϵ }
δ_6	{ $\epsilon,)$ }	{ $\epsilon,)$ }	{+, $\epsilon,)$ }	{+, ϵ }	{*, +, $\epsilon,)$ }
δ_7	{ $\epsilon,)$ }	{ $\epsilon,)$ }	{+, $\epsilon,)$ }	{+, $\epsilon,)$ }	{*, +, $\epsilon,)$ }

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

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 A-правила виду $A \rightarrow \omega_1 | \omega_2 | \dots | \omega_p$

- $First_1(\omega_i) \cap First_1(\omega_j) = \emptyset, i \neq j$
- якщо $\omega_i \Rightarrow^* \epsilon$, $First_1(\omega_j) \cap Follow_1(A) = \emptyset, 1 \leq j \leq p, j \neq i$

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

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

Побудуємо таблицю управління для наступної грамматики:

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

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

Правило	Номер правила	$First_1(\omega) \oplus_1 Follow_1(A)$
$S \rightarrow BA$	(1)	{(, a}
$A \rightarrow +BA$	(2)	{+}
$A \rightarrow \epsilon$	(3)	{ $\epsilon,)$ }
$B \rightarrow DC$	(4)	{(, a}
$C \rightarrow *DC$	(5)	{*}

$C \rightarrow \varepsilon$	(6)	{+, ε,)}
$D \rightarrow (S)$	(7)	{(}
$D \rightarrow a$	(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.

$A_i \backslash \Sigma$	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 $M(a,b)$:

Π_0 Прочитаємо поточну лексему з вхідного файлу, у стек магазинного автомата занесемо аксіому S.

....

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

- Якщо на вершині стека лексема $a_i = \langle \text{поточна лексема} \rangle$, то з вершини стека зняти a_i та прочитати нову поточну лексему.

- Якщо стек порожній та досягли кінця вхідного файлу, 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:

- $M(A, b) = \langle \text{номер правила} \rangle$ виду $A \rightarrow \omega_i$, такого, що $b \in First_1(\omega_i \cdot Follow_1(A))$

- $M(A, b) = \langle \text{ім'я програми} \rangle$ 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_scanner()`, яка при кожному зверненні 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 "mystack.h"
/* визначення структури стека синтаксичного 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_scanner(void); // функція виділення нової лексеми
extern void scanner_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 E-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, column, 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 роботи синтаксичного аналізатора */
      while (! NULL_STACK()) // поки стек не пустий
      { if (!ind) ind=1, pascal_scanner();
// А. 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));
              scanner_close(); return(0);
            }
// В. Обробка при умові, що на вершині нетермінал
            line0=index_elem(netname,numnet,COPY_STACK());
            if (lexema_code) column=index_elem(terminal,numtrm,lexema_code);
                else column=numtrm;
            if (upr= *(TABL_LL1_UPR+line0*(numtrm+1)+column))
                { 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));
            scanner_close(); return(0);
        } // кінець цикла обробки стека
// Стек порожній: перевіримо стан справ на вході
        if (! ind ) pascal_scanner(); scanner_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.

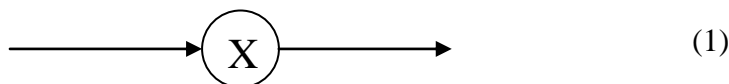
Оскільки у більшості випадків при визначенні синтаксису мови програмування ми користуємося множиною рекурсивних правил, то серед підпрограм, які будуються на основі правил грамматики, будуть і рекурсивні процедури (рекурсія буде як явна, так і неявна).

Сформулюємо правила побудови синтаксичного графа:

П₁. Кожен нетермінал з відповідною множиною породжуючих правил $A \rightarrow \omega_1 | \omega_2 | \dots | \omega_p, \omega_i \in (N \cup \Sigma)^*, i = 1 \dots p$ відображається в один синтаксичний граф. Отже, кількість синтаксичних графів рівна кількості нетерміналів грамматики G .

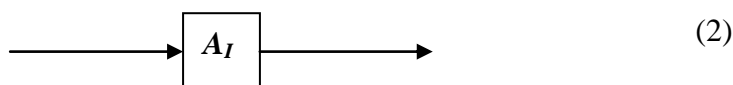
П₂. Для кожного елемента ланцюжка $\omega = \alpha_1 \alpha_2 \dots \alpha_p, \alpha_i \in N \cup \Sigma, i = 1 \dots p$ будуємо ребро синтаксичного графа та покажемо його таким чином що:

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



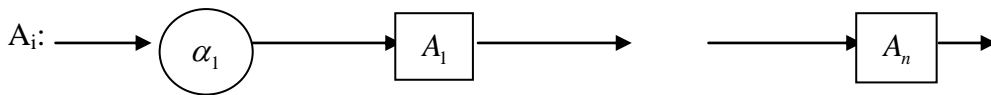
Мал.1.

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

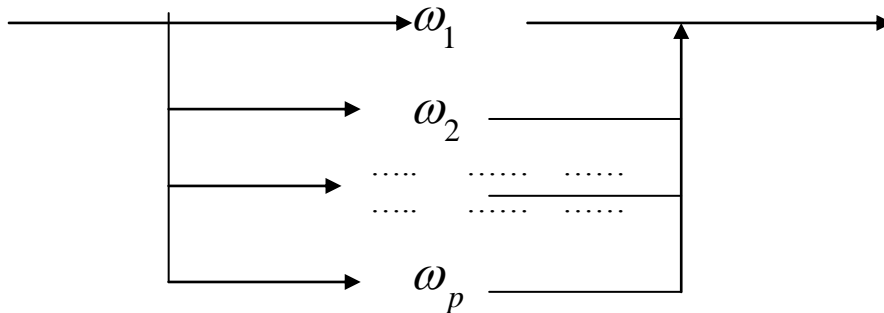


Мал.2 .

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



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



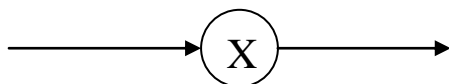
Мал. 3.

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

Якщо на основі грамматики 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 $A_i \rightarrow \omega_1 | \omega_2 | \dots | \omega_p$ з синтаксичною діаграмою виду (3) необхідне виконання наступної умови: множини $First_1(\omega_j) \oplus_1 Follow_1(A_i)$, $j = 1 \dots p$ 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

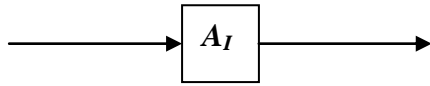


Мал.4.

відповідний фрагмент програми (наприклад, мовою C) 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

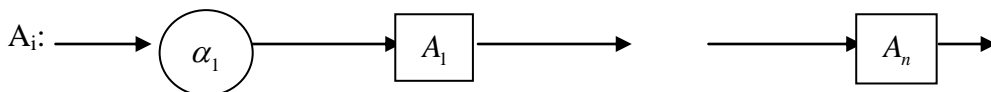


Мал.5.

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

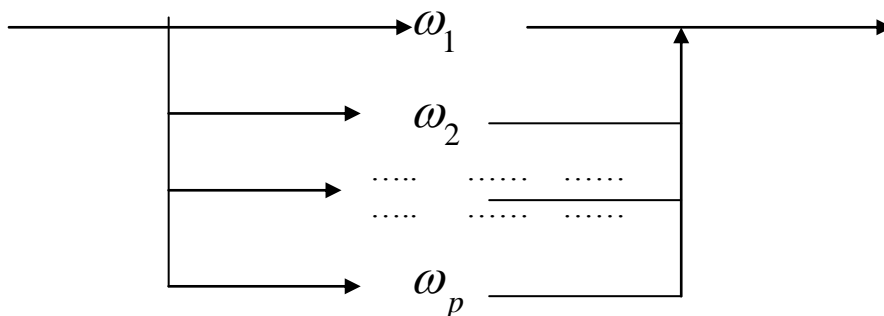


Мал.6.

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 A-правила, яка має вигляд:



Мал.7.

для кожного $\omega_i, i = 1 \dots p$ 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 $\omega_i, i = 1 \dots p$:

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

...

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

Оскільки за умовою $L_i \cap L_j = \emptyset, i \neq j$, 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_a1:
```

```
    case code_a1:
```

...

```
    case code_a1n1:// фрагмент програми для  $\omega_1$ 
```

...

```
    break;
```

```
    case code_a2:
```

```
    case code_a2:
```

...

```
    case code_a2n2:// фрагмент програми для  $\omega_2$ 
```

...

```
    break;
```

....

```
    case code_ap:
```

```
    case code_ap:
```

...

```
    case code_anp:// фрагмент програми для  $\omega_p$ 
```

...

```
    break;
```

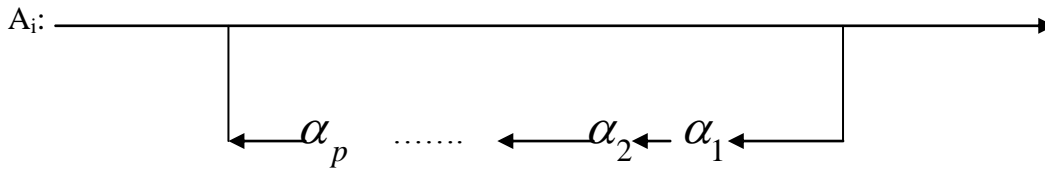
```
    default: error( );
```

```
  }
```

```
}//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 $A_i \rightarrow \alpha_1 \alpha_2 \dots \alpha_p A_i \mid \varepsilon$. Тоді синтаксична діаграма буде мати вигляд:



Мал. 8 .

Для вище наведеної синтаксичної діаграми відповідні множини будуть:

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

Відповідний фрагмент програми мовою C матиме вигляд:

```
extern int lexema_code;
extern char lexema_text[ ];
void Ai(void)
{ while (lexema_code == code_x1 //
        lexema_code == code_x1 //
        ...
        lexema_code == code_x1^n1 )
    { // фрагмент програми для ланцюжка  $\alpha_1\alpha_2\dots\alpha_p$ 
    ...
    }
} // кінець підпрограми для нетерміналу  $A_i$ 
```

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 ( );
  Ахіота_S ( ); // процедура, пов'язана з аксіомою граматики
}
```

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

Повернемося до умови, при якій граMATика G буде LL(k)-граматикою, а саме: для довільного виводу:

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

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

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

Означення. Множина

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

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

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

в інших випадках - невизначено.

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

в інших випадках - невизначено.

.....

$\Pi_n: \delta_n(S, A_i) = \delta_{n-1}(S, A_i) \cup \{L \mid S \rightarrow \varpi_1 A_i \varpi_2, L = First_k(w_2)\}$

в інших випадках - невизначено.

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

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:

$$A \rightarrow W_1 \mid W_2 \mid \dots \mid W_p$$

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

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

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

Для побудови синтаксичного аналізатора для $LL(k)$ -граматики ($k > 1$) необхідно побудувати множину таблиць, що забезпечать нам безтупиковий 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 граматичі G :

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

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, \dots, T_p визначаються так:

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

Наступні таблиці визначаються так:

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

Наступні таблиці T_1, T_2, \dots, T_p визначаються так:

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

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:

$$S \rightarrow abA \quad (1)$$

$$S \rightarrow \varepsilon \quad (2)$$

$$A \rightarrow Saa \quad (3)$$

$$A \rightarrow b \quad (4)$$

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

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

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

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

$$- S \rightarrow abA \quad (\text{правило 1})$$

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

$$- S \rightarrow \varepsilon \quad (\text{правило 2})$$

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

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:

$$- A \rightarrow Saa \quad (\text{правило 3})$$

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

$$- A \rightarrow b \quad (\text{правило 4})$$

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

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

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

$$- S \rightarrow abA \quad (\text{правило 1})$$

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

$$- S \rightarrow \varepsilon \quad (\text{правило 2})$$

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

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

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:

$$- A \rightarrow Saa \quad (\text{правило 3})$$

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

$$- A \rightarrow b \quad (\text{правило 4})$$

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

$T \setminus \Sigma^{*2}$	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:

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

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

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

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

П₀: Прочитати k лексем з вхідного файла програми (звичайно, інколи менше ніж k). В магазин занести таблицю T_0 .

....

П _{i} : - Якщо на вершині магазину знаходиться таблиця T_i , то елемент таблиці $M(T_i, <k\text{-вхідних лексем}>)$ визначає ланцюжок, який T_i заміщає на вершині магазину.

- Якщо на вершині магазину $a_i \in \Sigma$ і перша поточна лексема з k прочитаних лексем рівна a_i , то з вершини магазину 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.

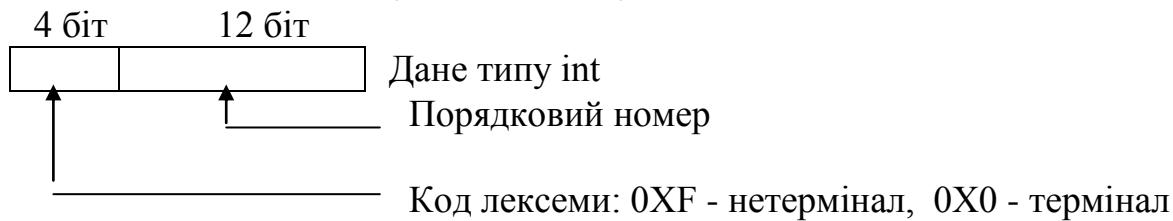
Граматики мови програмування визначаються множиною БНФ-правил, що записані в текстовому файлі. Кожне правило обов'язково починається з першої позиції рядка. Для зручності правило можна продовжити в наступних рядках, але не з першої позиції. Нетермінал граматики - це ланцюжок літер, який починається з символу $<$ та закінчується символом $>$, наприклад $<\text{програма}>$. 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 \backslash . Правило граматики в текстовому файлі записується так:

$<\text{програма}>$ $\text{program} (<\text{список параметрів}>) <\text{блок}> .$

В пам'яті ЕОМ грамика описується наступними структурами:

```
struct node { int *pd;           // посилка на продукцію
              int len;          // довжина продукції
              int teg;          // робоче поле
              struct node *next; // посилка на наступну продукцію
              struct fdata *cont_llk; // посилка на контекст для правила
};
```

Поле "довжина продукції" визначає кількість елементів продукції включаючи перший. Елементи продукції (нетермінали та термінали) закодовані цілочисловими даними, таким чином, що:



Для зберігання словарних множин $First_k$ та $Follow_k$ використовується наступна інформаційна структура:

```
struct fdata { struct fdata *fnext;      // посилка на наступний блок слів
               struct fdata *fnext1; // системне поле - не модифікувати
               int rdata[60];         // поле для слів
            };
```

Кожне слово - це послідовність лексем довжиною не більше k елементів. Якщо одного блоку для зберігання відповідної множини слів даних не достатньо, то в подальшому пам'ять виділяється динамічно. Признак кінця послідовності слів в одному блоці - (-1). Оскільки слова, які зберігаються в словарному блоці мають різну довжину, то на початку кожного слова стоїть його довжина.

Для доступу до закодованих в пам'яті ЕОМ даних надається ряд інтерфейсних функцій, наприклад:

- по коду лексеми отримати її текст:

```
char * NAME_ELEM(int);
```

- визначити кількість слів в словарній множині:

```
int calc_word(struct fdata p);
```

- отримати слово з індексом i (індексація починається з нуля) з словарної множини:

```
int * get_word(struct fdata p, int i);
```

Решта інтерфейсних функцій описані у файлі *mystand.h*.

Для виконання лабораторної роботи користувачеві надаються наступні бази даних:

```
extern struct fdata **ref_FIRST;      // масив посилки на множини  $First_k(A)$ 
extern struct fdata **ref_FOLLOW;    // масив посилки на множини  $Follow_k(S, A)$ 
extern struct flang **ref_LOCAL;     // масив посилки на множини  $Local_k(S, A)$ 
extern int LLk_LEN;                  // довжина  $llk$ -контексту
extern int *epsilon, num_epsilon;    // список  $\epsilon$ -нетерміналів
extern int *netname, numnet;         // список нетерміналів граматики
extern int *terminal, numtrm;       // список терміналів граматики
extern struct node *q_grammat;       // початок списку правил граматики
extern int LL1_USL;                  // відповідає за  $LL(1)$  - умову
extern char *TABL_LL1_UPR;           // таблиця управління  $LL(1)$  - аналізатора
```

Число рядків таблиці TABL_LL1_UPR рівне numnet, а число стовпчиків - numtrm+1. Останній стовпчик таблиці відповідає ерсилон-слову.

Відмітимо, що довжина масивів `ref_FIRST`, `ref_FOLLOW` та `ref_LOCAL` - це кількість нетерміналів граматики, а i -й елемент кожної множини асоціюється з i -м елементом списку нетерміналів.

5.1. Блок №1 лабораторних робіт

Мета: ознайомитися з методами зберігання КС-граматики в пам'яті ЕОМ для подальшої обробки та запрограмувати деякі функції доступу/ обробки наведених вище інформаційних структур.

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.

Вихідні дані:

- машинно-орієнтоване представлення множини КС-правил граматики;

2. Реалізувати алгоритм пошуку непродуктивних нетерміналів. Надрукувати список непродуктивних нетерміналів. Інтерфейс виклику для реалізації функції:

```
int prepr_std(q,r)
    struct node *q;           // список КС-правил
    struct dnode *r;         // початок хеш-таблиці
```

Функція повертає результат :

- 0 - коли непродуктивних нетерміналів немає,

- 1 - непродуктивні нетермінали в граматиці є.

3. Реалізувати алгоритм пошуку недосяжних нетерміналів. . Надрукувати список недосяжних нетерміналів. Інтерфейс виклику для реалізації функції:

```
int prepr_std(q,r)
    struct node *q;           // список КС-правил
    struct dnode *r;         // початок хеш-таблиці
```

Функція повертає результат :

- 0 - коли недосяжних нетерміналів немає,

- 1 - недосяжні нетермінали в граматиці є.

4. Реалізувати алгоритм пошуку ліворекурсивних нетерміналів. . . Надрукувати список ліворекурсивних нетерміналів. Інтерфейс виклику для реалізації функції:

```
int prepr_std(q,r)
    struct node *q;           // список КС-правил
    struct dnode *r;         // початок хеш-таблиці
```

Функція повертає результат :

- 0 - коли ліворекурсивних нетерміналів немає,

- 1 - ліворекурсивні нетермінали в граматиці є.

5. Реалізувати алгоритм пошуку праворекурсивних нетерміналів. Надрукувати список праворекурсивних нетерміналів. Інтерфейс виклику для реалізації функції:

```
int prepr_std(q,r)
    struct node *q;           // список КС-правил
    struct dnode *r;         // початок хеш-таблиці
```

Функція повертає результат :

- 0 - коли праворекурсивних нетерміналів немає,

- 1 - праворекурсивні нетермінали в граматиці є.

6. Розробити та реалізувати алгоритм пошуку різних виводів виду $A \Rightarrow^* w$ мінімальної довжини, тобто мінімальних ліворекурсивних виводів. Надрукувати послідовність правил, які дають такий результат. Інтерфейс виклику для реалізації функції:

```
int prepr_std(q,r)
    struct node *q;           // список КС-правил
    struct dnode *r;        // початок хеш-таблиці
```

Функція повертає результат :

- 0 - коли ліворекурсивних нетерміналів немає,

- 1 - ліворекурсивні нетермінали в граматиці є.

7. Розробити та реалізувати алгоритм пошуку різних виводів виду $A \Rightarrow^* wA$ мінімальної довжини, тобто мінімальних праворекурсивних виводів. Надрукувати послідовність правил, які дають такий результат. Інтерфейс виклику для реалізації функції:

```
int prepr_std(q,r)
    struct node *q;           // список КС-правил
    struct dnode *r;        // початок хеш-таблиці
```

Функція повертає результат :

- 0 - коли праворекурсивних нетерміналів немає,

- 1 - праворекурсивні нетермінали в граматиці є.

8. Реалізувати алгоритм пошуку множини $Follow_k(A)$ для нетерміналів граматики. Надрукувати множини $Follow_k(A)$ для кожного нетерміналу. Інтерфейс виклику для реалізації функції:

```
int prepr_std(q,r)
    struct node *q;           // список КС-правил
    struct dnode *r;        // початок хеш-таблиці
```

9. Реалізувати алгоритм пошуку множини $First_k(A)$ для нетерміналів граматики. Надрукувати множини $First_k(A)$ для кожного нетерміналу. Інтерфейс виклику для реалізації функції:

```
int prepr_std(q,r)
    struct node *q;           // список КС-правил
    struct dnode *r;        // початок хеш-таблиці
```

10. Побудувати $LL(1)$ -граматику для мови програмування С. Якщо виконати $LL(1)$ -умову неможливо, то спробуйте для деяких правил запропонувати $LL(1)$ -умову. Інтерфейс виклику для реалізації функції:

```
int prepr_std(q,r)
    struct node *q;           // список КС-правил
    struct dnode *r;        // початок хеш-таблиці
```

11. Реалізуйте допоміжні функції, які використовуються для побудови множин $First_k(A)$.

- функція побудови слова довжиною не більше k символів:

```
int newword(mult,dwor,wrk,len)
    struct fdata **mult; // масив посилки на скінчені мови
    short int *dwor;     // масив поточна перестановка індексів слів
```



```

short int wrk[]; // масив під результуюче слово
int len; // число множин, які перемножаться
- функція побудови нової перестановки індексів для словарних множин:
int setword(mult,max_dwor,dwor,len)

```

```

struct fdata **mult; // масив посилки на скінчені мови
int *max_dwor; // кількість слів в кожній словарній множині
int *dwor; // поточна перестановка індексів
int len; // число множин, які перемножаться

```

12. Реалізувати функцію, котра на основі контекстів $First_k(A)$ для кожного правила виду $A_i \rightarrow W$ обчислить значення $First_k(W)$ - для кожної правої частини правила. Результат занести в поле cont_llk для кожної продукції. Інтерфейс виклику для реалізації функції:

```

int prepr_std(q,r)
struct node *q; // список КС-правил
struct dnode *r; // початок хеш-таблиці

```

```

int prepr_std(q,r)

```

13. Реалізувати функцію, котра на основі контекстів $First_k(A)$ и $Follow_k(A)$ для нетерміналів обчислить значення $First_k(W) +_k Follow_k(A)$ для кожної правої частини продукції виду $A \rightarrow W$. Результат занести в поле cont_llk для кожної продукції. Інтерфейс виклику для реалізації функції:

```

int prepr_std(q,r)
struct node *q; // список КС-правил
struct dnode *r; // початок хеш-таблиці

```

14. Реалізувати функцію, котра на основі контекстів $First_k(W) +_k Follow_k(A)$ для кожної правої частини продукції виду $A \rightarrow W$ визначить, чи є граматики сильною $LL(k)$ -граматикою, зокрема $LL(1)$ -граматикою. Контексти по кожному правилу знаходяться в полі cont_llk. Інтерфейс виклику для реалізації функції:

```

int prepr_std(q,r)
struct node *q; // список КС-правил
struct dnode *r; // початок хеш-таблиці

```

15. (це приклад). Реалізувати алгоритм пошуку EPSILON-нетерміналів. Надрукувати список EPSILON-нетерміналів, а також правила граматики, які дають вивід виду: $A \Rightarrow *$. Інтерфейс виклику для реалізації функції:

```

int prepr_std(q,r)
struct node *q; // список КС-правил
struct dnode *r; // початок хеш-таблиці

```

Функція повертає результат :

- 0 - коли EPSILON-нетерміналів немає,
- 1 - коли EPSILON-нетермінали в граматиці є.

5.2. Блок №2 лабораторних робіт

Реалізація $LL(k)$ - синтаксичного аналізатора для відомих мов програмування.

1. Реалізувати функцію, котра для граматики обчислить множини $Local_k(S, A)$, такі що:

$LOCAL(S, A) = \{ L \mid S \Rightarrow^* xAw, L = FIRST_k(w) \text{ при відповідних } x \text{ и } w \}$

Інтерфейс виклику для реалізації функції:

```
int prepr_std(q,r)
    struct node *q;           // список КС-правил
    struct dnode *r;         // початок хеш-таблиці
```

2. Реалізувати функцію, котра на основі множин $Local_k(S, A)$ перевіряє, чи є граMATика $LL(k)$ -граматикою ($k > 1$). Інтерфейс виклику для реалізації функції:

```
int prepr_std(q,r)
    struct node *q;           // список КС-правил
    struct dnode *r;         // початок хеш-таблиці
```

3. Реалізувати функцію, котра на основі контексту $First_k(W) +_k Follow_k(A)$ для кожної правої частини продукції виду $A \rightarrow W$ (результат знаходиться в полі `cont_llk` для кожної продукції) будує таблицю управління для $LL(1)$ -синтаксичного аналізатора. Інтерфейс виклику для реалізації функції:

```
int prepr_std(q,r)
    struct node *q;           // список КС-правил
    struct dnode *r;         // початок хеш-таблиці
```

4. Реалізувати синтаксичний аналізатор для мови PL/0 методом магазинного автомата. Для управління роботою аналізатора скористайтеся таблицею управління для $LL(1)$ -синтаксичного аналізатора. Інтерфейс виклику для реалізації функції:

```
int prepr_std(q,r)
    struct node *q;           // список КС-правил
    struct dnode *r;         // початок хеш-таблиці
```

Як допоміжну функцію вводу лексем з текстового файлу PL/0-програми використайте лексичний аналізатор з PL/0 з наступною специфікацією:

```
extern char scanner_lexema[ ];           // текст виділеної лексеми
extern int lexema_code;                 // код лексем
int scanner_pl0(void).
```

Текст лексеми заноситься в поле `scanner_lexema`. Функція `scanner_pl0` повертає код виділеної лексеми. Якщо досягли кінця вхідного файлу, то функція `scanner_pl0` повертає значення 0(нуль). По закінченню роботи синтаксичного аналізатора необхідно закрити вхідний файл при допомозі функції `scanner_close()`.

5. Реалізувати синтаксичний аналізатор для мови PASCAL методом магазинного автомата. Для управління роботою аналізатора скористайтеся таблицею управління для $LL(1)$ -синтаксичного аналізатора. Інтерфейс виклику для реалізації функції:

```
int prepr_std(q,r)
    struct node *q;           // список КС-правил
    struct dnode *r;         // початок хеш-таблиці
```

Як допоміжну функцію вводу лексем з текстового файлу Pascal -програми використайте лексичний аналізатор з Pascal з наступною специфікацією:

```
extern char scanner_lexema[ ];           // текст виділеної лексеми
```

```
extern int lexema_code;      // код лексем
    int pascal_scanner(void).
```

Текст лексеми заноситься в поле `scanner_lexema`. Функція `pascal_scanner()` повертає код виділеної лексеми в поле `lexema_code`. Якщо досягли кінця вхідного файлу, то функція `pascal_scanner ()` повертає значення 0(нуль). По закінченню роботи синтаксичного аналізатора необхідно закрити вхідний файл при допомозі функції `scanner_close()`.

6. Реалізувати синтаксичний аналізатор для мови PL/0 методом рекурсивного спуску. Для управління роботою аналізатора скористайтесь таблицею управління для LL(1)-синтаксичного аналізатора. Інтерфейс виклику для реалізації функції:

```
int prepr_std(q,r)
    struct node *q;          // список КС-правил
    struct dnode *r;        // початок хеш-таблиці
```

Як допоміжну функцію вводу лексем з текстового файлу PL/0-програми використайте лексичний аналізатор з PL/0 з наступною специфікацією:

```
extern char scanner_lexema[ ];      // текст виділеної лексеми
extern int lexema_code;             // код лексем
    int scanner_pl0(void).
```

Текст лексеми заноситься в поле `scanner_lexema`. Функція `scanner_pl0` повертає код виділеної лексеми. Якщо досягли кінця вхідного файлу, то функція `scanner_pl0` повертає значення 0(нуль). По закінченню роботи синтаксичного аналізатора необхідно закрити вхідний файл при допомозі функції `scanner_close()`.

7. Реалізувати синтаксичний аналізатор для мови Pascal методом рекурсивного спуску. Для управління роботою аналізатора скористайтесь таблицею управління для LL(1)-синтаксичного аналізатора. Інтерфейс виклику для реалізації функції:

```
int prepr_std(q,r)
    struct node *q;          // список КС-правил
    struct dnode *r;        // початок хеш-таблиці
```

Як допоміжну функцію вводу лексем з текстового файлу Pascal-програми використайте лексичний аналізатор з Pascal з наступною специфікацією:

```
extern char scanner_lexema[ ];      // текст виділеної лексеми
extern int lexema_code;             // код лексем
    int pascal_scanner(void).
```

Текст лексеми заноситься в поле `scanner_lexema`. Функція `pascal_scanner()` повертає код виділеної лексеми в поле `lexema_code`. Якщо досягли кінця вхідного файлу, то функція `pascal_scanner ()` повертає значення 0(нуль). По закінченню роботи синтаксичного аналізатора необхідно закрити вхідний файл при допомозі функції `scanner_close()`.

8. Реалізувати методом магазинного синтаксичний аналізатор мови програмування С. Врахувати, що на вхід синтаксичного аналізатора поступає програма після обробки препроцесором. Інтерфейс виклику для реалізації функції:

```
int prepr_std(q,r)
    struct node *q;          // список КС-правил
    struct dnode *r;        // початок хеш-таблиці
```

9. Реалізувати методом магазинного синтаксичний аналізатор мови програмування TURBO PASCAL 5.xx. Врахувати, що на вхід синтаксичного

аналізатора поступає програма після обробки препроцесором. Інтерфейс виклику для реалізації функції:

```
int prepr_std(q,r)
    struct node *q;           // список КС-правил
    struct dnode *r;         // початок хеш-таблиці
```

5.3. Блок №3 лабораторних робіт

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

1. Побудувати LL(k)-граматику (k=1 или k=2) для мови програмування Delphi. Реалізувати синтаксичний аналізатор мови програмування Delphi.
2. Скористайтесь інструментальним комплексом LEX/ YACC та реалізуйте синтаксичний аналізатор мови програмування Turbo Pascal 5.xx.
3. Скористайтесь інструментальним комплексом LEX/ YACC та реалізуйте синтаксичний аналізатор мови програмування C.
4. Скористайтесь інструментальним комплексом LEX/ YACC та реалізуйте синтаксичний аналізатор мови програмування C++.
5. Реалізуйте препроцесор мови програмування C++.
- 6 Реалізуйте препроцесор мови програмування C.

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., Цейтлин Г.Е., Ющенко Е.Л. Алгебра, языки, программирование. 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.