МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Ю.І. Стативка

ФОРМАЛЬНІ МОВИ

Основні концепти і представлення

Навчальний посібник

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для студентів спеціальності
121 Інженерія програмного забезпечення

Електронне мережне навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2023

Рецензент: *Жаріков Е. В.*, д.т.н., доцент, завідувач кафедри інформатики та програмної інженерії ФІОТ Національного технічного університету України «Київський політехнічний інститут» імені Ігоря Сікорського

Відповідальний редактор: *Свинчук О.В.*, к.ф.-м.н., доцент

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського (протокол № 8 від 02.06.2023 р.) за поданням Вченої ради Навчально-наукового інституту атомної та теплової енергетики (протокол № 13 від 29.05.2023 р.)

В навчальному посібнику представлені елементи теорії формальних мов, генеративних граматик та розпізнавачів. Описані актуальні нотації та способи подання синтаксису і семантики мов програмування, розглянуті відповідні сучасні інструментальні засоби.

Навчальний посібник призначений для студентів спеціальності 121 «Інженерія програмного забезпечення».

Реєстр № НП 22/23-756 Обсяг 4,1 авт. арк.
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Проспект Перемоги, 37, м. Київ, 03056
http://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

Зміст

Вступ

Формальні мови становлять концептуальну основу мов програмування та інших засобів і складових інформаційних технологій, як-то мов опису форматів файлів, протоколів взаємодії, конфігураційних файлів, документів, віртуальних світів тощо. Тому знання теоретичних основ та наявність досвіду роботи з формальними мовами важливі для розробників програмного забезпечення, а для розробників мов програмування, компіляторів та інтерпретаторів вони – необхідні.

Навчальний посібник призначений для підтримки лекційної складової односеместрової дисципліни "Основи розробки трансляторів", в межах якої кожен студент має створити власну імперативну мову загального призначення з наперед заданими характеристиками та реалізувати її (побудувати компілятор).

Таке завдання, зокрема, передбачає не тільки знання основ теорії формальних мов та уміння маніпулювати граматиками, але також і розуміння важливості опису семантики програмних конструкцій та знання сучасних підходів до специфікації мов програмування.

Навчальний посібник складається з двох розділів, перший з яких містить, на думку автора, мінімальний набір фундаментальних понять та фактів, необхідний для професійної діяльності кожного розробника програмного забезпечення.

У другому розділі розглядаються символьні і графічні метамови представлення лексики та синтаксису мов програмування. Окремо розглядається поняття синтаксичного дерева, з'ясовуються причини неоднозначності його використання. Висвітлюються питання семантики програмних конструкцій та способів її представлення. Нарешті, наводиться типовий зміст специфікації мов програмування, з посиланнями як на сучасні і потужні `C\#, Java, Python`, так і на старі компактні `Algol 60` та `Pascal`.

Скрізь, де це можливо, автор віддавав перевагу роз'ясненням і змістовним прикладам перед формальним поданням. Вибір між цими альтернативами спричинений обмеженістю часу, доступному в межах курсу. Водночас, для зацікавленого читача надаються посилання на літературні джерела: на класичні праці класиків інформатики з фундаментальних питань, на відносно нові, але апробовані, з прикладних питань, і нарешті, на найновіші програмні інструменти та специфікації, доступні для вільного використання.

Для поглибленого вивчення тем, розглянутих у першому розділі, традиційно і обґрунтовано рекомендуються книги [1, 2, 3]. Перша з них, легендарна "Книга

дракона"[1] [1], висвітлює питання, пов'язані з побудовою компіляторів, і тому мови та автомати розглядаються, здебільшого, утилітарно. Натомість, у книзі [2], мови та автомати є, фактично, основним предметом розгляду і описані докладно. Нарешті, "Теорія синтаксичного аналізу, перекладу і компіляції" [3], де класична, але актуальна, теорія формальних мов висвітлюється надзвичайно повно. Знайомство з цим джерелом не є обов'язковим для студентів, проте фахова діяльність з побудови компіляторів без нього навряд чи можлива.

При підготовці цього навчального посібника автор намагався, крім матеріалу необхідного і мінімально достатнього, представити також інформацію про загальний "ландшафт", яка потребує не вивчення, але ознайомлення. Мотивація такого підходу – дефрагментація поля знань читача. Серед таких питань, зокрема, лінійно обмежені автомати і машина Тьюринга, варіанти розширеної нотації Бекуса-Наура, інші нотації з власними назвами, підходи до формального представлення семантики тощо.

Розділ `Зміст` організовано так, аби він відображав не тільки організацію тексту, але і організацію предметної області. Всі посилання у тексті – інтерактивні ("клікабельні").

Автор, який аж ніяк не є верстальником, вважає, що тільки використання системи LaTeX дозволило підготувати цей навчальний посібник до видання і зазначає свою вдячність спільноті TeX Users Group за підтримку цієї видавничої системи.

[1] "Dragon Book"

Розділ 1. Концептуальні основи

1.1. Формальні мови

Для розгляду формальних мов дамо, насамперед, визначення низки необхідних понять. Формальні мови, як і природні, передбачають можливість подання текстів у формі послідовності символів певного алфавіту.

Алфавітом називають непорожню множину неподільних символів (букв, літер) і позначають великими літерами грецького алфавіту чи латиниці (можливо з індексами), як-то Σ, A, B, V.

Наприклад, алфавіти $\Sigma_1 = \{a, b, c\}$, $A_1 = \{00, 01, 10, 11\}$ і $V_1 = \{0, 1\}$ містять 3, 4 і 2 символи відповідно. В алфавіті A_1 кожен з чотирьох символів вважається неподільним за визначенням.

Ланцюжок (слово) над алфавітом Σ -– це послідовність літер цього алфавіту. Для позначення ланцюжків часто використовують літери грецького алфавіту, наприклад, α, β чи ω, або літери другої половини латинського алфавіту, наприклад p, s, t, x, y тощо.

Довжина слова α визначається як кількість літер у цьому слові. Позначається як $|\alpha|$, або як $\ell(\alpha)$. Наприклад, $|bc| = 2$ над алфавітом Σ_1, $|010011| = 3$ над алфавітом A_1 і $|010011| = 6$ над алфавітом V_1.

Порожній ланцюжок позначають через ε або λ. Очевидно, що довжина порожнього слова $|\varepsilon| = 0$.

Конкатенацією слів α та β є слово $\gamma = \alpha\beta$, тобто до слова α справа дописують β. Так, якщо $\alpha = ab$ та $\beta = cd$, то їх конкатенація $\alpha\beta = abcd$. Конкатенація – некомутативна операція, оскільки, $\alpha\beta \neq \beta\alpha$, адже $abcd \neq cdab$.

Очевидним є результат конкатенації довільного слова x з порожнім рядком: $\varepsilon x = x\varepsilon = x$.

Якщо слово $w = xy$, то кажуть, що x – префікс (голова), а y – суфікс (хвіст) слова w. Префікс x називають власним (правильним) префіксом (правильною головою) w, якщо $x \neq \varepsilon$ і $x \neq w$. Відповідно y називають власним (правильним) суфіксом (хвостом) слова w, якщо $y \neq \varepsilon$ і $y \neq w$. І взагалі, довільний підрядок s рядка (слова, ланцюжка) $w = \alpha s \beta$ є його власним підрядком, якщо $s \neq \varepsilon$ і $s \neq w$.

Для спрощення та формалізації роботи з довгими словами застосовують степінь ланцюжка, де показник степеня вказує на кількість повторень підрядка у певному місці слова. Так слово $aaaaa$ зручно позначити як a^5, $abababab$ – як $(ab)^4$, а $abababbbaaaabab$ – як $(ab)^3b^2a^3(ab)^2$. Взагалі, степені довільного слова α означають:

$\alpha^0 = \varepsilon$
$\alpha^1 = \alpha$
$\alpha^2 = \alpha\alpha$
$\alpha^3 = \alpha\alpha\alpha$
...
$\alpha^n = \alpha^{n-1}\alpha$

Над довільними множинами ланцюжків можна визначити операцію конкатенації (добутку над операцією конкатенації): $AB = \{xy \mid x \in A, y \in B\}$.

Наприклад, для $A = \{m, n\}, B = \{0, 1\}$ добуток $AB = \{m0, m1, n0, n1\}$.

Оскільки алфавіт – це множина, то всі ланцюжки довжини n над алфавітом можна представити як його n-ий степінь. Нехай алфавіт $\Sigma = \{a, b\}$. Тоді всі слова довжини 3 над Σ можна представити у формі

$$\Sigma^3 = \Sigma\Sigma\Sigma = \{aaa, aab, aba, abb, baa, bab, bba, bbb\}.$$

І взагалі
$\Sigma^0 = \{\varepsilon\}$ – єдине порожнє слово,
$\Sigma^1 = \Sigma = \{a, b\}$ – усі можливі слова довжини 1,
$\Sigma^2 = \Sigma\Sigma = \{aa, ab, ba, bb\}$ – усі можливі слова довжини 2,
$\Sigma^3 = \Sigma\Sigma\Sigma = \{aaa, aab, aba, abb, baa, bab, bba, bbb\}$ – усі слова довжини 3,
...
$\Sigma^n = \Sigma^{n-1}\Sigma$ – усі можливі слова довжини n.

Множину всіх ланцюжків над алфавітом Σ називають замиканням Кліні (зірка Кліні, ітерація Кліні, а то й просто замикання) та позначають

$$\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^i = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cdots \cup \Sigma^n \ldots$$

Замикання Кліні алфавіту $\Sigma = \{a, b\}$ – множина усіх можливих слів, довжиною від нуля до нескінченності:

$$\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb \ldots, aaaa, \ldots\}.$$

Позитивним замиканням алфавіту (плюс Кліні) називають множину

$$\Sigma^+ = \bigcup_{i=1}^{\infty} \Sigma^i = \Sigma^1 \cup \Sigma^2 \cdots \cup \Sigma^n \ldots$$

Так, для $\Sigma = \{a, b\}$ позитивне замикання Σ^+ – множина усіх можливих над Σ слів, довжиною від одного до нескінченності:

$$\Sigma^+ = \{a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb \ldots, aaaa, \ldots\}.$$

З означень безпосередньо випливає $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$.

Мовою L над алфавітом Σ називають деяку підмножину замикання Кліні алфавіту $L \subset \Sigma^*$.

Так, замикання алфавіту $\Sigma = \{a\}$ – це нескінченна множина слів довільної довжини $\Sigma^* = \{\varepsilon, a, aa, aaa, aaaa, \ldots\}$. Визначимо мову L як множину слів, довжина кожного з яких – непарна, тобто $L = \{a, aaa, aaaaa, \ldots\}$.

До формальних мов широко застосовують як звичайні множинні операції – об'єднання, перетин та доповнення, так і специфічні: конкатенація –

$L_1L_2 = \{xy \mid x \in L_1, y \in L_2\}$; ітерація або Кліні зірочка (*) (оператор замикання); Кліні плюс ($^+$); заміна (підстановка); морфізми та зворотні морфізми; обернення (реверс); перемішування тощо.

Для означення мови використовують запис у формі

$$L = \{w \mid \text{ – опис властивостей } w\},$$

які читаються так: "мова L – це множина слів w, таких, що мають вказані після вертикальної риски властивості".

Для наведеного вище прикладу мова L може бути визначена, наприклад, так: $L = \{w \mid \ell(w) \text{ – непарне число}\}$, або, з більш формальним визначенням довжини слова: $L = \{w \mid w \in \Sigma^*, |w| = 2n+1, n \geq 0\}$. В останньому записі кома означає кон'юнкцію, тобто визначення може бути прочитане так: "мова L – це множина слів w із множини Σ^* з довжиною, що визначається виразом $2n+1$, де n – невід'ємне ціле число, або число нуль".

Отже, для визначення формальної мови L необхідно:

1. обрати алфавіт та назвати його, наприклад, Σ;
2. побудувати[1] замикання алфавіту Σ^*;
3. вказати вимоги до властивостей слів з Σ^*, які входять до означуваної мови.

Такий підхід дозволяє перевірити, чи справді кожне слово даного тексту (наприклад, комп'ютерної програми) написане означеною формальною мовою (вказаною мовою програмування). Таке завдання виконується з використанням розпізнавачів – автоматів (автоматів з магазинною пам'яттю, або, у найпростіших випадках – скінченних автоматів чи, еквівалентних їм, регулярних виразів).

Альтернативний підхід полягає у використанні формальних граматик, які містять правила, що дозволяють згенерувати тільки такі ланцюжки символів, які гарантовано належать означеній мові.

Тож для реалізації вказаних підходів до визначення мов, розглянемо такі абстрактні математичні системи, як:

1. регулярні вирази;
2. формальні (генеративні) граматики;
3. розпізнавачі (автомати).

1.2. Регулярні вирази

Регулярні вирази використовують для визначення регулярних мов. Тому спочатку дамо визначення регулярої мови (регулярної множини), а потім – регулярного виразу.

Рекурсивно регулярна мова над алфавітом Σ може бути визначена так:

[1] Замикання алфавіту – нескінченна множина, тому говорячи "побудувати", маємо на увазі "вміти будувати", або "розуміти як побудувати"

1. порожня мова $\varnothing$ є регулярною мовою;
2. $\{\varepsilon\}$ є регулярною мовою над алфавітом Σ (вона містить єдине слово – порожнє);
3. для кожного $a \in \Sigma$ множина $\{a\}$ є регулярною мовою (вона містить єдине слово – a);
4. якщо R та S – регулярні мови над алфавітом Σ, то регулярними також є мови:
   - $R \cup S$ – об'єднання регулярних мов;
   - RS – конкатенація регулярних мов;
   - R^* – замикання Кліні регулярної мови (довільної над алфавітом Σ);
5. інших регулярних мов над алфавітом Σ не існує.

Регулярні вирази – засіб позначення регулярних мов, прийнятий для представлення та формальних маніпуляцій над регулярними мовами, визначаються так:

1. $\varnothing$ – регулярний вираз, що позначає порожню мову $\varnothing$;
2. $\{\varepsilon\}$ – регулярний вираз, що позначає регулярну мову $\{\varepsilon\}$;
3. $\{\mathbf{a}\}$ –регулярний вираз, що позначає регулярну мову $\{a\}$;
4. якщо $\mathbf{r}$ та $\mathbf{s}$ – регулярні вирази, що позначають, відповідно, регулярні мови R та S над алфавітом Σ, то:
   - $\mathbf{r} + \mathbf{s}$ – регулярний вираз [1], що позначає об'єднання регулярних мов $R \cup S$;
   - $\mathbf{rs}$ – регулярний вираз [2], що позначає конкатенацію регулярних мов RS;
   - $\mathbf{r}^*$ – регулярний вираз, що позначає замикання Кліні регулярної мови R^*;
   - $(\mathbf{r})$ – регулярний вираз, що позначає ту ж регулярну мову R, що й регулярний вираз $\mathbf{r}$ (тобто зайві дужки можуть бути опущені);
5. інших регулярних виразів над алфавітом Σ не існує.

Наведемо приклади регулярних виразів, що визначають відповідні регулярні мови над певним алфавітом:

- ab позначає мову над алфавітом $\Sigma = \{a, b\}$, що містить одне слово $\{ab\}$;

[1] Замість оператора + також використовується символ | (вертикальна риска).

[2] Оператор конкатенації, зазвичай, опускають, але інколи застосовують явне позначення, наприклад символ . (крапка).

- $a+b$ позначає мову над алфавітом $\Sigma = \{a, b\}$, що містить два слова $\{a, b\}$;
- $a + (b^*)$ позначає мову над алфавітом $\Sigma = \{a, b\}$:
  $\{a\} \cup \{b\}^* = \{a, \varepsilon, b, bb, bbb, \dots\}$;
- $aa(a + b)^*$ позначає мову над алфавітом $\Sigma = \{a, b\}$, кожне слово якої має префікс aa, а решта слова – нуль або більше символів, кожен з яких – це a чи b;
- $a(ba)^* + b(ab)^*$ позначає мову без порожнього слова над алфавітом $\Sigma = \{a, b\}$, кожне слово якої починається і закінчується одним і тим же символом, і в якому символи a і b почергово змінюють один одного;
- $(0 + 1)(0 + 1 + de)^*$ позначає мову над алфавітом $\Sigma = \{0, 1, d, e\}$, кожне слово якої починається з 0 або 1, а решта слова містить нуль чи більше ланцюжків, кожний з яких – це 0, 1 або de.

Одна й та ж регулярна мова може бути представлена різними регулярними виразами. Справді, наприклад, $a\varepsilon b$, εab, та $ab\varepsilon$ позначають одну й ту ж мову $\{ab\}$ над алфавітом $\Sigma = \{a, b\}$. Два регулярні вирази α та β, що позначають одну й ту ж регулярну мову, називають еквівалентними та пишуть $\alpha = \beta$. Еквівалентність регулярних виразів з'ясовується (доводиться) через доведення рівності відповідних їх мов.

Для регулярних виразів α, β та γ справедливі, зокрема, такі еквівалентності:

- комутативність та асоціативність оператора альтернативи $+$:
  $\alpha + \beta = \beta + \alpha$,
  $\alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma$;
- дистрибутивність (зліва та справа) конкатенації відносно альтернативи $+$:
  $\alpha(\beta + \gamma) = \alpha\beta + \alpha\gamma$,
  $(\alpha + \beta)\gamma = \alpha\gamma + \beta\gamma$;
- порожнє слово ε є одиницею відносно конкатенації:
  $\alpha\varepsilon = \varepsilon\alpha = \alpha$;
- порожнє слово ε завжди міститься в ітерації:
  $\alpha^* = (\alpha + \varepsilon)^*$;
- порожня мова $\varnothing$ є нулем відносно конкатенації та нулем відносно альтернативи $+$:
  $\alpha\varnothing = \varnothing\alpha = \varnothing$,
  $\alpha + \varnothing = \alpha$;
- ідемпотентність ітерації $*$:
  $(\alpha^*)^* = \alpha^*$;
- позитивне замикання:
  $\alpha^+ = \alpha\alpha^*$.

Метамова представлення регулярних виразів

Правила запису регулярних виразів фактично є метамовою – мовою для опису іншої мови. Для усунення неоднозначності при опусканні чи відсутності дужок, необхідно враховувати пріоритет операторів (всі вони – лівоасоціативні), поданих далі у порядку його зменшення:

- унарний постфіксний оператор $*$ – ітерація (замикання Кліні), має найвищий пріоритет;
- бінарний оператор конкатенації, як правило не має явного позначення і має проміжне значення пріоритету щодо $*$ та $+$;
- бінарний оператор $+$ – альтернативи (об'єднання мов), має найнижчий пріоритет.

Часте застосування регулярних виразів, зокрема і в практиці програмування, призвели до значного розширення мінімальної метамови регулярних виразів та виникнення численних діалектів. Типовими, зокрема, є використання таких операторів та метасимволів:

- унарний постфіксний оператор позитивного замикання $^+$ [1]:
  $\alpha^+ = \alpha\alpha^*$ – один або більше екземплярів α. Пріоритет дорівнює пріоритету оператора $*$;
- унарний постфіксний оператор ?:
  $\alpha? = \varepsilon + \alpha$ – нуль або один екземпляр α. Пріоритет дорівнює пріоритету оператора $*$;
- квадратні дужки (метасимволи) [ ] – позначають клас символів (один з перелічених):
  $a_0 + a_1 + \ldots + a_n$, де $a_i \in \Sigma$ представляють у формі $[a_0 a_1 \ldots a_n]$, наприклад $0+1+2+3+4+5+6+7+8+9+A+B+C+D+E+F$ можна представити як $[0123456789ABCDEF]$, або, якщо визначені стандартні послідовності літер та цифр, у формі $[0-9A-F]$.

Граф регулярного виразу

Довільний регулярний вираз може бути представлений у формі орієнтованого поміченого графа, дуги якого помічені символами з $\Sigma \cup \varepsilon$, де Σ – алфавіт регулярної мови.

Побудову поміченого орієнтованого графа регулярного виразу r представимо у вигляді набору (пронумерованих) правил з графічними ілюстраціями до них, див. рис. 1.1–1.4. Для побудови графа регулярного виразу r:

1. подають його у формі двох вершин – стартової та заключної (подвійне коло), з дугою, поміченою символом r:

[1] Тоді для позначення оператора альтернативи використовується символ | (вертикальна риска).

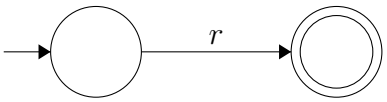

Рис. 1.1. До правила 1 побудови графа

2. виконують заміну кожної дуги з альтернативою $r_1 + r_2$ на дві дуги між тими ж вузлами з позначками r_1та r_2:

Рис. 1.2. До правила 2 побудови графа

3. виконують заміну кожної дуги з конкатенацією r_1r_2 на дві дуги r_1 і r_2 з вершиною між ними:

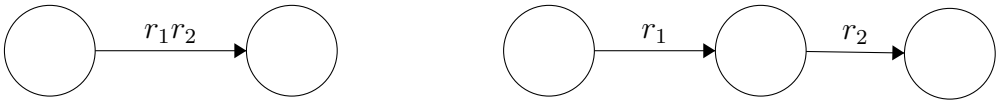

Рис. 1.3. До правила 3 побудови графа

4. виконують заміну кожної дуги з ітерацією r_1^* на три дуги та вершину з позначками ε, r_1 та ε відповідно:

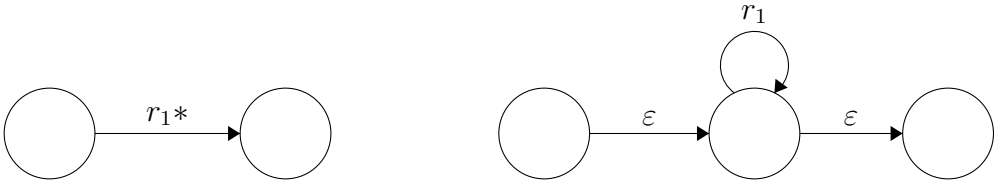

Рис. 1.4. До правила 4 побудови графа

5. виконують стягування кожної дуги ε, якщо вона є єдиною вихідною дугою некінцевої вершини, або єдиною вхідною дугою нестартової вершини, в одну вершину.

Скористаємось наведеними правилами для побудови графа регулярного виразу $r_1 = a(ba)^*(b + \varepsilon) + b(ab)^*(a + \varepsilon)$, подаючи цей процес покроково, див. рис. 1.5–1.11:

1. стартова – заключна вершини:

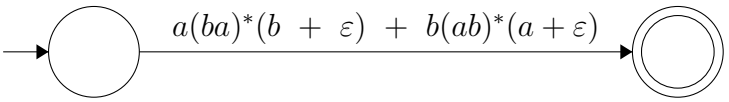

Рис. 1.5. За правилом 1

2. альтернатива:

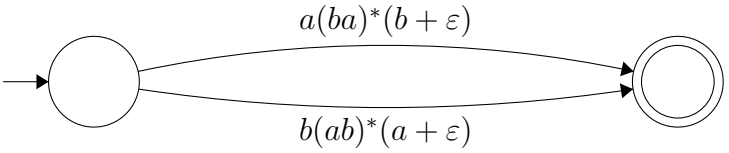

Рис. 1.6. За правилом 2

3. конкатенація:

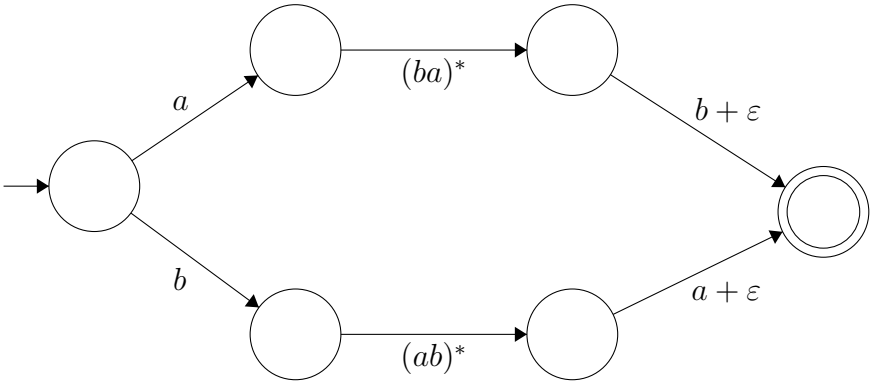

Рис. 1.7. За правилом 3

4. альтернатива:

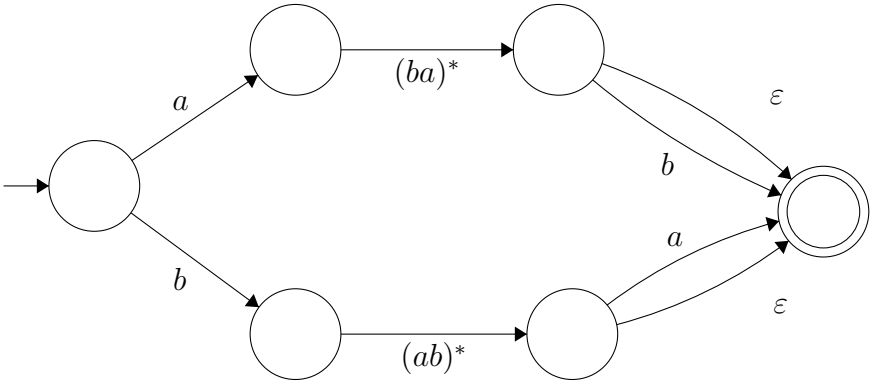

Рис. 1.8. За правилом 2

5. замикання Кліні у кожній гілці:

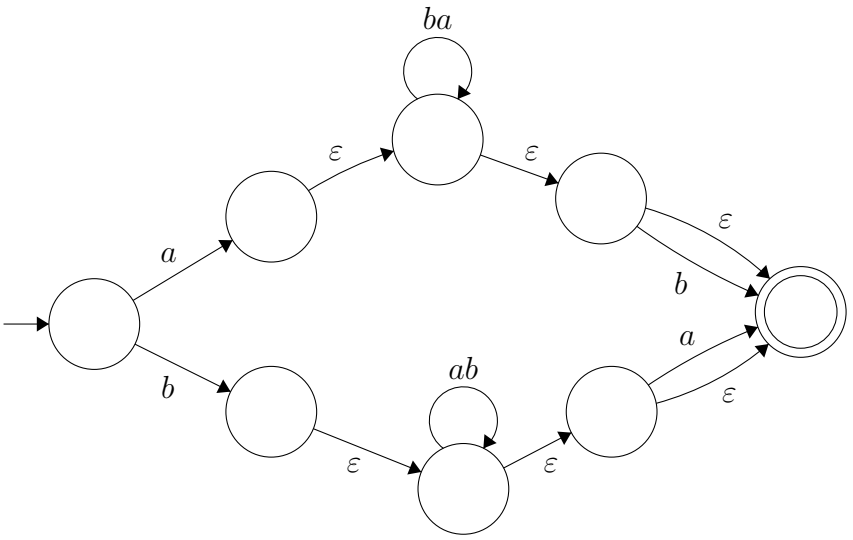

Рис. 1.9. За правилом 4

6. конкатенація:

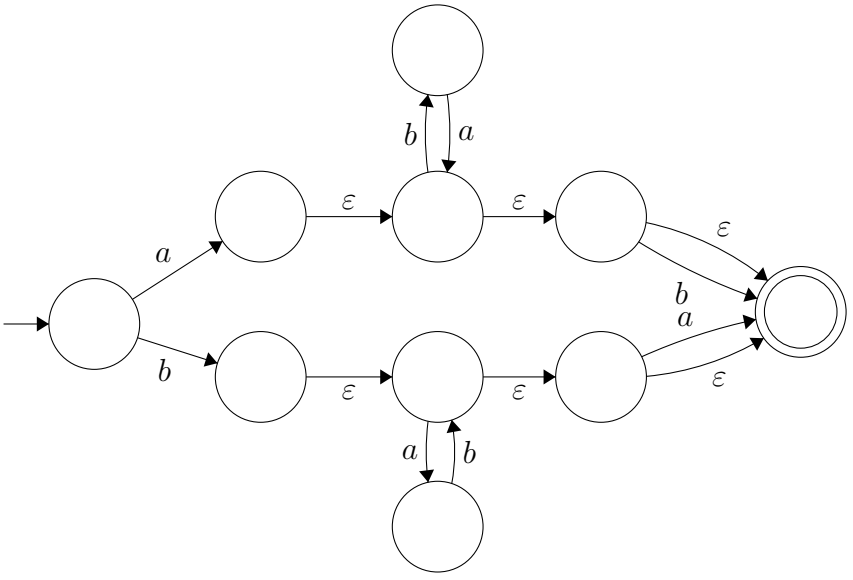

Рис. 1.10. За правилом 3

7. стягування ε-дуг:

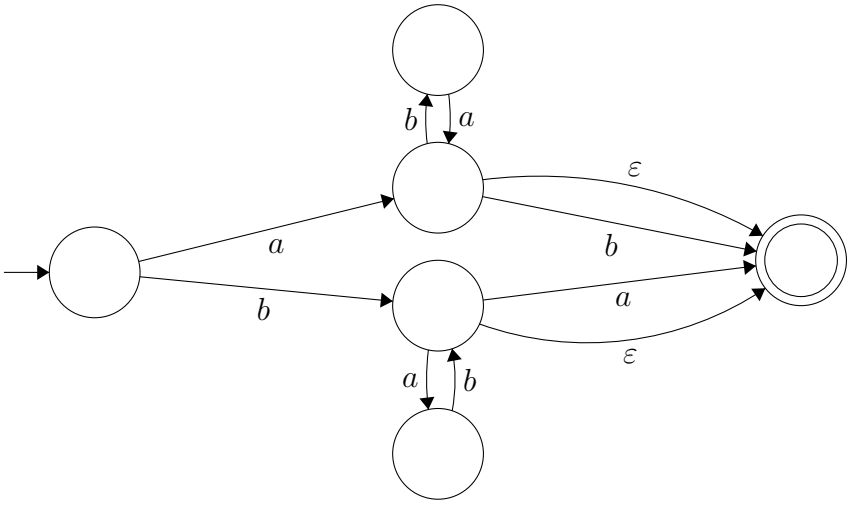

Рис. 1.11. За правилом 5

Побудований тут граф регулярного виразу є графом переходів недетермінованого скінченного автомата, який визначає ту ж мову, що і регулярний вираз, див. розд. 1.4.1.

1.3. Формальні граматики

Формальною породжувальною граматикою (генеративною граматикою, граматикою з фразовою структурою, граматикою з переписуванням) називають кортеж

$$\Gamma = (N, T, P, S)^1,$$

де N – множина нетермінальних символів (нетерміналів, змінних);
T – множина термінальних символів (терміналів, основних символів);
$N \cap T = \varnothing$;
P – множина правил виведення (продукцій, правил підстановки) $\{(\alpha, \beta)\}$;
S – початковий символ (стартовий символ, аксіома).

Продукції $\{(\alpha, \beta)\}$ множини P подаються у формі $\alpha \to \beta$, причому $\alpha \in (N \cup T)^* N (N \cup T)^*$ і $\beta \in (N \cup T)^*$, звідки $P \subseteq (N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$

Зауважимо, що алфавіт граматики і алфавіт мови завжди розрізняються. Алфавітом граматики Γ є $N \cup T$, оскільки в граматиці використовуються як нетермінальні, так і термінальні символи. У мові, яку визначає граматика – тільки термінали, отже алфавітом мови є множина T.

Далі, якщо не сказано інше, використовуються такі символи для позначень:

- великі латинські літери $A, B, C, \ldots$ – для нетерміналів. Літера S, якщо вона є, вважається стартовим символом;

[1] Символ Γ читається як "гамма".

- великі латинські літери $X, Y, \ldots$ – для довільних окремих символів (терміналів чи нетерміналів);
- малі латинські літери з початку алфавіту $a, b, c, \ldots$ та цифри $0, 1, \ldots$ – для терміналів;
- грецькі $\alpha, \beta, \gamma, \ldots$ – для ланцюжків, до складу яких можуть входити термінали та нетермінали;
- літери u, v, w, x, y – для ланцюжків термінальних символів;
- усі вказані літери можуть використовуватись з нижніми індексами, або зі штрихом, наприклад, $A', B_{12}, c_1, \gamma_3, u_2, v''$ тощо.

Приклад граматики: $\Gamma = (\{A, S\}, \{a, b\}, \{S \to aAb, A \to aAb, A \to \varepsilon\}, S)$. Як бачимо тут: множина нетерміналів $N = \{A, S\}$, множина терміналів $T = \{a, b\}$, множина правил $P = \{S \to aAb, A \to aAb, A \to \varepsilon\}$, стартовий символ – S.

Часто, із-за значної кількості правил, граматику зручніше подати у такій формі:

$$\Gamma = (\{A, S\}, \{a, b\}, P, S),$$

де P містить продукції:

$$S \to aAb$$
$$A \to aAb$$
$$A \to \varepsilon$$

Наведені продукції читають, наприклад, так "стартовий нетермінал S визначається як (слово) aAb, нетермінал A визначається як (слово) aAb або як порожній рядок ε". Замість слів "визначається як (слово)" кажуть також "є за означенням", "породжує (слово)" тощо.

Множину продукцій P можна записати компактніше, використовуючи оператор альтернативи | (вертикальна риска):

$$S \to aAb$$
$$A \to aAb \mid \varepsilon$$

Метамова представлення граматик

Наведені правила запису граматик фактично є метамовою – мовою для опису іншої мови (мови представлення граматик). Крім наведених правил позначення, вона містить такі оператори:

- стрілочка ($\to$), розділяє праву та ліву частину кожного правила;
- альтернативи, позначається як (|), або розуміється між правими частинами продукцій з однаковими лівими частинами;
- конкатенації, його символ опускається, у продукції $S \to aAb$ він передбачається між символами a, A та b;

Пріоритети операторів зростають від найменшого (стрілочка) до найвищого (конкатенація) через проміжне значення (альтернатива).

1.3.1. Виведення

Нехай граматика Γ містить продукцію $\alpha \to \beta$ і слово $\xi_1 = \gamma_1\alpha\gamma_2$, де слова $\gamma_1, \gamma_2 \in (N \cup T)^*$. Застосування продукції $\alpha \to \beta$ до слова ξ_1 породжує слово $\xi_2 = \gamma_1\beta\gamma_2$ (застосування зводиться до заміни α на β в ξ_1). Тоді кажуть, що слово ξ_2 безпосередньо виводиться зі слова ξ_1, і позначають як $\xi_1 \Rightarrow \xi_2$.

Якщо $\xi_0, \xi_1, \ldots, \xi_n$ – слова над $(N \cup T)^*$ та $\xi_0 \Rightarrow \xi_1 \Rightarrow \cdots \Rightarrow \xi_{n-1} \Rightarrow \xi_n$, то кажуть, що ξ_n виводиться з ξ_0, та записують як $\xi_0 \overset{*}{\Rightarrow} \xi_n$.

Послідовність кроків для отримання ξ_n з ξ_0 називають виведенням, а n – довжиною (кількістю кроків) виведення.

Виведення $\xi_0 \Rightarrow \xi_1 \Rightarrow \cdots \Rightarrow \xi_{n-1} \Rightarrow \xi_n$ називають повним, якщо слово ξ_n не містить нетерміналів. Саме слово ξ_n при цьому називають реченням. Довільне слово ξ_i виведення називають його сентенційною формою.

Тепер можна сформулювати ще одне, еквівалентне вже наведеним, визначення мови: мовою $L(\Gamma)$, породжуваною граматикою $\Gamma = (N, T, P, S)$, називають множину всіх термінальних ланцюжків (речень), які виводяться з аксіоми S:

$$L(\Gamma) = \{\xi \mid \xi \in T^*, S \overset{*}{\Rightarrow} \xi\}.$$

Приклад. Виведемо кілька слів, заданих граматикою

$$\Gamma = (\{A, S\}, \{0, 1\}, \{S \to 0A1, A \to 0A1, A \to \varepsilon\}, S). \tag{1.1}$$

Для зручності перенумеруємо продукції множини P:

(1) $S \to 0A1$
(2) $A \to 0A1 \mid \varepsilon$

Виконаємо наступні кроки для виведення одного зі слів:

1. $S \overset{1}{\Rightarrow} 0A1$ – за правилом (1);
2. $0A1 \overset{2.1}{\Rightarrow} 00A11$ – за правилом (2.1), тобто першою альтернативою правила (2);
3. $00A11 \overset{2.2}{\Rightarrow} 00\varepsilon11$ – за правилом (2.2), тобто другою альтернативою правила (2);
4. ε – порожній рядок, тож після підстановки його значення в ланцюжок $00\varepsilon11$ (тобто викреслювання) маємо, насамкінець, 0011.

Отже, слово 0011 належить мові $L(\Gamma)$, оскільки $S \overset{*}{\Rightarrow} 0011$.

Надалі цю послідовність кроків представлятимемо більш компактно:
$S \overset{1}{\Rightarrow} 0A1 \overset{2.1}{\Rightarrow} 00A11 \overset{2.2}{\Rightarrow} 00\varepsilon11 = 0011$,
або, опускаючи тривіальний крок заміни ε його значенням, у формі:
$S \overset{1}{\Rightarrow} 0A1 \overset{2.1}{\Rightarrow} 00A11 \overset{2.2}{\Rightarrow} 0011$.

Виведемо ще кілька слів:
$S \overset{1}{\Rightarrow} 0A1 \overset{2.2}{\Rightarrow} 0\varepsilon1 = 01$
$S \overset{1}{\Rightarrow} 0A1 \overset{2.1}{\Rightarrow} 00A11 \overset{2.2}{\Rightarrow} 00\varepsilon11 = 0011$

$S \overset{1}{\Rightarrow} 0A1 \overset{2.1}{\Rightarrow} 00A11 \overset{2.1}{\Rightarrow} 000A111 \overset{2.2}{\Rightarrow} 000\varepsilon 111 = 000111,$
$S \overset{1}{\Rightarrow} 0A1 \overset{2.1}{\Rightarrow} 00A11 \overset{2.1}{\Rightarrow} 000A111 \overset{2.1}{\Rightarrow} 0000A1111 \overset{2.2}{\Rightarrow} 0000\varepsilon 1111 = 00001111.$

Легко помітити, що при кожному застосуванні продукції (2.1) префікс поповнюється одним нулем, суфікс – одиницею, між якими знову з'являється нетермінал A. Застосовування ж продукції (2.2) приводить до слова, яке є реченням, тобто у його складі не лишається нетермінального символу і процес виведення слова завершується.

Отже, заданій граматикою (1.1) мові $L(\Gamma)$ можна дати опис:

1. природною мовою (тут – українською): мова $L(\Gamma)$ містить непорожні слова, кожне з яких має префікс, що складається з нулів (0) та суфікс – з одиниць (1), і довжина префікса дорівнює довжині суфікса;

2. формальниий (мовою теорії множин):
   $L(\Gamma) = \{w_1 w_2 \mid w_1 \in \{0\}^+, w_2 \in \{1\}^+, |w_1| = |w_2|\}$;

3. формальний (з використанням алгебраїчних формул):
   $L(\Gamma) = \{0^n 1^n \mid n \geq 1\}$.

Дерево виведення

Повертаючись до виведення, зауважимо, що покрокове представлення процесу виведення у формі дерева, називають деревом виведення. На рис. 1.12 наведено виведення слова 000111 у граматиці 1.1, де, для зручності, показано також продукції, застосовані на кожному кроці виведення.

Конкатенація термінальних вузлів (листків) дерева виведення дає слово, що належить мові. Для внутрішніх та термінальних вузлів дерева використовують різні графічні позначення, тут термінальні вузли (листки) позначені символом у колі, нетермінальні – просто символом. Порожній рядок, оскільки нічого не породжує, як і термінальні символи, позначено колом.

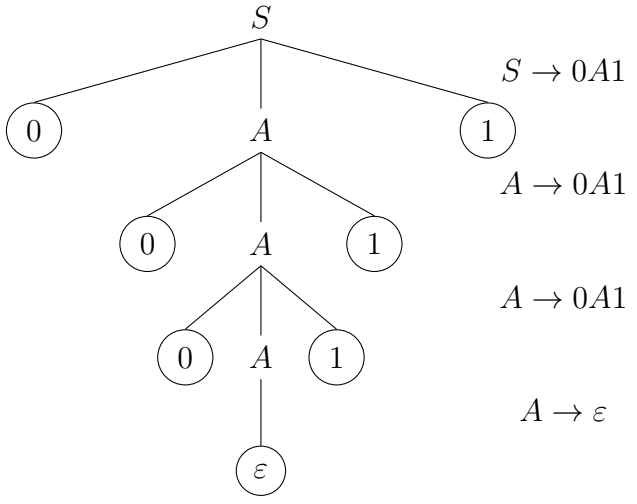

Рис. 1.12. Дерево виведення слова 000111

Одне й те ж дерево виведення може відповідати кільком виведенням, які відрізняються тільки порядком, в якому обирають нетермінали слова для застосування продукцій граматики. Щоб забезпечити одноманітність виведення при побудові одного й того ж дерева,застосовують, наприклад, ліве виведення – це коли спершу обирають найлівіший нетермінал. Аналогічно визначається праве виведення. При виведенні обов'язково зазначають, яке саме виведення використовується.

Якщо слово $w \in L(\Gamma)$ має два або більше різних дерев виведення, то граматику Γ називають неоднозначною.

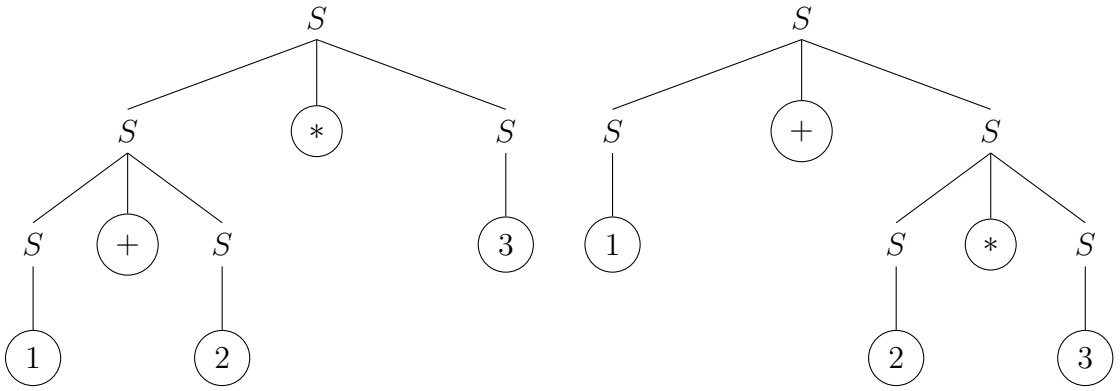

Рис. 1.13. Дерева виведення слова $1 + 2 * 3$ у граматиці Γ_1.

На рис. 1.13 представлено два різних виведення слова $1 + 2 * 3$ у граматиці $\Gamma_1 = (\{S\}, \{1, 2, 3, +, *\}, \{S \to S + S \mid S * S \mid 1 \mid 2 \mid 3\}, S)$. Вона містить єдиний нетермінал S і тому вибір дерева визначається тільки порядком застосування продукцій: дерево а) на рис. 1.13 якщо першою застосовується продукція $S \to S * S$, або дерево б), рис. 1.13, якщо спочатку – $S \to S + S$.

Очевидним кроком для усунення неоднозначності граматики є введення додаткового нетермінала, який забезпечить правильний порядок застосування продукцій у, наприклад, лівому виведенні. Тоді продукції $S \to S + S \mid S * S$ можна подати як $S \to S + A$ та $A \to A * N$, що забезпечить застосування спочатку продукції $S \to S + A$, і тільки потім $A \to A * N$. Таким чином, трансформовану однозначну граматику Γ_2, еквівалентну граматиці Γ_1, можна записати як $\Gamma_2 = (\{S\}, \{1, 2, 3, +, *\}, \{S \to S + A \mid 1 \mid 2 \mid 3, A \to A * N \mid 1 \mid 2 \mid 3, N \to 1 \mid 2 \mid 3\}, S)$

Ліве виведення слова $1 + 2 * 3$ у однозначній граматиці Γ_2:
$S \overset{1.1}{\Rightarrow} S + A \overset{1.2}{\Rightarrow} 1 + A \overset{2.1}{\Rightarrow} 1 + A \overset{2.1}{\Rightarrow} 1 + A * N \overset{2.3}{\Rightarrow} 1 + 2 * N \overset{3.3}{\Rightarrow} 1 + 2 * 3$ у формі дерева виведення представлено на рис. 1.14.

1.3.2. Структура слова та правила граматики

Побудова граматики мови – процес неформальний, оскільки не існує загального алгоритму для побудови граматики довільної, наперед визначеної мови. Тому граматики будують, користуючись евристичними правилами, відомими з досвіду виконання таких завдань. Деякі з таких евристик, виглядають так:

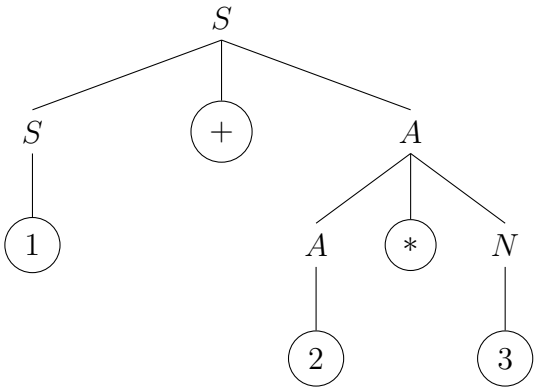

Рис. 1.14. Дерево виведення $1 + 2 * 3$ у граматиці Γ_2

1. Якщо слово містить тільки речення w, то використовують правило $A \to w$;

2. Якщо слово починається з речення w, то використовують правило $A \to wA$;

3. Якщо слово закінчується реченням w, то використовують правило $A \to Aw$;

4. Якщо слово починається з речення w_1 та закінчується реченням w_2, то використовують правило $A \to w_1 A w_2$;

5. Якщо в середині слова має зустрічатись речення w, то використовують правило $A \to BwC$;

6. Якщо слово містить одне чи більше повторень ланцюжка w , то використовують правила
   $A \to Aw \mid w$ – ліворекурсивне правило, або
   $A \to wA \mid w$ – праворекурсивне правило;

7. Якщо у слові є чергування ланцюжків w_1 та w_2, то використовують правила
   $A \to w_1 B \mid w_1$
   $B \to w_2 A \mid w_2$

Приклад. Побудувати граматику мови, кожне слово якої починається з двох літер "a" та закінчується літерою "b", між якими знаходиться нуль чи більше літер "c".

Як це видно з опису, множина терміналів $T = \{a, b, c\}$.

Приймемо, що S – стартовий символ.

Зважаючи на те, що кожне слово має починатися з двох літер "a" та закінчуватись літерою "b", скористаємось правилом 4 для побудови першої продукції у

формі $S \to aaAb$. Оскільки кількість літер "c" може дорівнювати нулю, побудуємо правило $A \to \varepsilon$. Для випадку, коли літера "c" повторюється один чи більше разів, згідно з правилом 6, оберемо праворекурсивну продукцію $A \to cA \mid c$. Тоді всі альтернативи для виведення з нетермінала A можна подати як $A \to cA \mid c \mid \varepsilon$. Ці три альтернативи еквівалентні двом альтернативам $A \to cA \mid \varepsilon$, позаяк породжують однакові множини слів: $\{\varepsilon, c, cc, ccc, \dots\}$. (Справді, адже послідовне застосування альтернатив $A \to cA$ та $A \to \varepsilon$ еквівалентне альтернативі $A \to c$)

Отже, множина нетерміналів $= \{A, S\}$.

Тоді граматику даної мови можна представити так:

$$\Gamma = (\{A, S\}, \{a, b, c\}, \{S \to aaAb, A \to cA \mid \varepsilon\}, S).$$

1.3.3. Використання граматик

Оскільки формальні граматики здатні моделювати більшу частину довільної мови програмування, то виникає кілька типових завдань практичного характеру, зокрема:

1. Визначити, чи належить ланцюжок певній мові.
2. Знайти опис мови, заданої граматикою.
3. Побудувати граматику за описом мови.

Визначити, чи належить ланцюжок певній мові

Очевидно, треба за описом мови перевірити, чи має даний ланцюжок (слово) властивості, що висуваються до усіх слів мови.

Слово ξ належить мові з граматикою Γ, якщо вдається побудувати його виведення у граматиці даної мови Γ, тобто якщо $S \overset{*}{\Rightarrow} \xi$, де S – стартовий символ граматики. Або, що те ж саме, якщо вдається побудувати дерево виведення даного слова.

Завдання визначення належності ланцюжка певній мові може бути вирішене з використанням розпізнавачів (автоматів) – абстрактних систем, опис яких наведено у розділі 1.4, після розгляду питання про класифікацію мов. Така послідовність розгляду вмотивована залежністю типу розпізнавального автомата від типу мови.

Для вирішення двох інших завдань використовують такі неформальні рекомендації.

Знайти опис мови, заданої граматикою

Для виконання такого завдання треба:

1. Вивести за правилами граматики якусь кількість речень.
2. Зауважити закономірність, притаману ланцюжкам мови (а вона завжди є).

3. Дати короткий опис поміченій закономірності (словами природної мови та, якщо можливо, – формальний).

4. Впевнитись, що за правилами граматики можуть бути виведені й інші ланцюжки, що відповідають знайденому опису, (бажано – "ті і тільки ті" замість "й інші", проте це не завжди можливо).

Приклад виконання такого завдання для граматики

$$\Gamma = (\{A, S\}, \{0, 1\}, \{S \to 0A1, A \to 0A1, A \to \varepsilon\}, S)$$

див. на стор. 18.

1. Виведемо кілька речень:
   $S \overset{1}{\Rightarrow} 0A1 \overset{2.2}{\Rightarrow} 0\varepsilon 1 = 01$
   $S \overset{1}{\Rightarrow} 0A1 \overset{2.1}{\Rightarrow} 00A11 \overset{2.2}{\Rightarrow} 00\varepsilon 11 = 0011$
   $S \overset{1}{\Rightarrow} 0A1 \overset{2.1}{\Rightarrow} 00A11 \overset{2.1}{\Rightarrow} 000A111 \overset{2.2}{\Rightarrow} 000\varepsilon 111 = 000111,$
   $S \overset{1}{\Rightarrow} 0A1 \overset{2.1}{\Rightarrow} 00A11 \overset{2.1}{\Rightarrow} 000A111 \overset{2.2}{\Rightarrow} 0000\varepsilon 1111 = 00001111.$

2. Бачимо, що всі розглянуті термінальні ланцюжки (речення) починаються з символа нуль (0) та закінчуються символом одиниця (1). Всі нулі знаходяться на початку слів, усі одиниці – в кінці, нулі та одиниці не перемішуються. Порожнього слова у цій граматиці згенерувати не можна. У розглянутих словах кількість нулів у префіксі завжди дорівнює кількості одиниць у суфіксі.

3. Дати короткий опис поміченій закономірності (словами природної мови та, якщо можливо, – формальний):

   - природномовний: мова $L(\Gamma)$ містить непорожні слова, префікс яких складається з нулів (0), суфікс – з одиниць (1), і довжина префікса дорівнює довжині суфікса;
   - формальний:
     $L(\Gamma) = \{w_1 w_2 \mid w_1 \in \{0\}^+, w_2 \in \{1\}^+, |w_1| = |w_2|\}$ або
     $L(\Gamma) = \{0^n 1^n \mid n \geq 1\}$.

4. Впевнитись, що за правилами граматики можуть бути виведені й інші ланцюжки, що відповідають знайденому опису.
   Продукція 1.1 породжує слово $0A1$, з якого можна отримати найкоротше речення за допомогою продукції 2.2, а саме 01. Отже $L(\Gamma)$ – ε-вільна мова.
   Продукція 2.1 завжди збільшує на один символ (0) префікс та на один символ (1) суфікс з нетерміналом A між ними, що передбачає подальші кроки виведення. Отже, n застосувань продукції 2.1 збільшить префікс на n нулів та суфікс на n одиниць, а загальна довжина термінальних символів слова – $(n + 1 + n + 1)$. Подальше застосування продукції 2.2 приводить до термінального слова (речення) з префіксом та суфіксом довжини $n + 1$ кожний. Отже, $L(\Gamma)$ – мова, у якій довжина префікса кожного речення дорівнює довжині його суфікса.

Побудувати граматику за описом мови

Будувати граматику можна у такій послідовності :

1. Визначити множину терміналів (алфавіт мови).
2. Побудувати якусь кількість речень у відповідності до опису.
3. Записати правила граматики для виведення слів мови, див. розд. 1.3.2 .
4. Впевнитись, що за правилами побудованої граматики виводяться й інші ланцюжки, що відповідають даному опису мови, (бажано – "ті і тільки ті" замість "й інші", що, як вже зазначалось, не завжди можливо);
5. Записати граматику.

Приклад виконання такого завдання для мови, кожне слово якої починається з двох літер "a" та закінчується літерою "b", між якими знаходиться нуль чи більше літер "c" наведено на стор. 21.

1. Множина терміналів $T = \{a, b, c\}$.
2. Приклади речень мови: $aab, aacb, aaccb, aacccccccb$.
3. Записати правила граматики для виведення слів мови, див. розд. 1.3.2: $\{S \to aaAb, A \to cA \mid c \mid \varepsilon\}$.
4. Впевнитись, що побудована граматика відповідає опису мови:
   Стартовий символ породжує єдине слово $aaAb$.
   Наступне застосування правила $A \to \varepsilon$ породжує речення aab, – нуль літер c.
   Застосування до слова $aaAb$ продукції $A \to cA$ породжує слово $aacAb$ з – однією літерою c та нетерміналом A. Наступні кроки виведення приводять до слова $aacb$ при застосуванні продукції $A \to \varepsilon$ або до $aac^{n+1}b$ при наступному n-кратному застосуванні продукції $A \to cA$ та завершальною $A \to \varepsilon$. Отже, граматика породжує слова з префіксом aa, суфіксом b та літерою c між ними у кількості нуль, або більше.
5. Граматика даної мови:
   $\Gamma = (\{A, S\}, \{a, b, c\}, \{S \to aaAb, A \to cA \mid \varepsilon\}, S)$

1.3.4. Класифікація граматик

Формальні граматики, як правило, класифікують за структурою продукцій та, можливо, деякими іншими обмеженнями.

Регулярна граматика (праволінійна, автоматна, скінченно-автоматна) має продукції виду $A \to \alpha \mid \alpha B$, де $A, B \in N$, $\alpha \in T^*$.

Безконтекстна граматика (контекстно вільна, КВ) має продукції виду $A \to \alpha$, де $A \in N, \alpha \in (N \cup T)^*$.

Контекстна граматика (контекстно залежна, КЗ) має продукції виду $\alpha \to \beta$, де $\alpha = \gamma_1 A \gamma_2$, $\beta = \delta_1 \gamma_1 \omega \gamma_2 \delta_2$, $\gamma_1, \gamma_2, \delta_1, \delta_2 \in (N \cup T)^*$, $A \in N$, $\omega \in (N \cup T)^+$, а γ_1 і γ_2 називають контекстом, – лівим і правим відповідно.

Інакше можна сказати, що контекстно залежні граматики містять правила виду $\gamma_1 A \gamma_2 \to \delta_1 \gamma_1 \omega \gamma_2 \delta_2$, тобто ω можна вивести з ланцюжка сентенційної форми, тільки якщо A знаходиться між ланцюжками γ_1 та γ_2 .

Виокремлюють також граматики без обмежень, які не містять обмежень на правила виведення, за тим винятком, що у лівій частині не може бути порожнього рядка ε.

Для класифікації граматик за типами продукцій широко застосовують так звану ієрархію Чомскі[1], яка передбачає такі чотири їх типи:

- граматики типу 0 – граматики без обмежень;
- граматики типу 1 – контекстно залежні граматики;
- граматики типу 2 – безконтекстні граматики;
- граматики типу 3 – регулярні граматики.

Мови, згенеровані граматиками вказаних типів, знаходяться у відношенні включення: $L_{\text{тип3}} \subset L_{\text{тип2}} \subset L_{\text{тип1}} \subset L_{\text{тип0}}$.

Породжувальні граматики можуть використовуватись як для генерування, так і для розпізнавання мов. Найчастіше використовують граматики типів 2 та 3 для розпізнавання формальних і, частково, природних мов.

1.3.5. Трансформація граматик

Для формальних систем типовою є ситуація, коли один і той же об'єкт, залишаючись змістовно незмінним, може приймати безліч зовнішніх форм (представлень). Така ситуація знайома нам, наприклад з алгебри, коли потрібно спростити арифметичний чи алгебраїчний вираз. Так, довільний поліном другого степеня у рівнянні від однієї змінної спочатку приводять (шляхом еквівалентних перетворень) до стандартної форми $ax^2 + bx + c = 0$, а вже потім застосовують алгоритм його розв'язання.

Оскільки довільна формальна мова може бути представлена багатьма різними за формою, проте еквівалентними, граматиками, то виникає питання про стандартну, кажуть також – нормальну, форму граматики та засоби гарантовано еквівалентних перетворень граматик.

Нормальними формами контекстно вільних граматик є нормальні форми Чомскі, Грейбах та операторна. Кожна контекстно вільна граматика може бути ефективно приведена до однієї з названих нормальних форм.

До нормальної форми Куроди приводять контекстно залежні граматики.

[1] Ім'я автора класифікації Noam Chomsky транслітерують також як Ноам Хомський.

Нагадаємо, що граматики $\Gamma(T, N, P, S)$ та $\Gamma'(T', N', P', S')$ називають еквівалентними, якщо вони визначають одну й ту ж мову, $L(\Gamma) = L(\Gamma')$. Еквівалентним називають таке претворення (трансформацю) граматики $\Gamma(T, N, P, S)$ у $\Gamma'(T', N', P', S')$, що $L(\Gamma) = L(\Gamma')$. Далі, якщо не вказано інше, перетворення вважається еквівалентним.

Різноманітні трансформації застосовують з метою:

- спрощення – з граматики можуть бути усунуті певні правила та символи;
- зміни виду та/або структури граматики – граматика може бути поповнена певними правилами та нетермінальними символами.

Усунення ε-продукцій

Продукцію виду $A \to \varepsilon$, називають ε-продукцією, а нетермінал A при цьому – ε-породжуючим, або кажуть, що нетермінал A породжує порожнє слово ε. Взагалі, якщо $A \overset{*}{\Rightarrow} \varepsilon$, то A – ε-породжуючий нетермінал.

Довільний нетермінал, що визначається продукцією $A \to B_0 B_1 \dots B_n$, може бути ε-породжуючим тільки за умови, що всі B_i – нетермінали і кожний з них – ε-породжуючий. Якщо граматика не містить ε-продукції, то в ній немає ε-породжуючих нетерміналів.

При вилученні ε-продукції $X \to \varepsilon$ необхідно кожну продукцію, що визначається через X, записати як дві продукції – у початковій формі (випадок $X \neq \varepsilon$) та у формі без X (випадок $X = \varepsilon$).

Наприклад у граматиці

$$\begin{aligned} &A \to BCD \\ &C \to F \mid G \mid \varepsilon \\ &B \to w_1 \\ &D \to w_2 \\ &F \to w_3 \\ &G \to w_4 \end{aligned}$$

де $w_1, w_2, w_3, w_4 \in T^+$ – речення, нетермінали B, D, F і G не є ε-породжуючими нетерміналами, а C – ε-породжуючий нетермінал. У випадку $C \to \varepsilon$ продукцію $A \to BCD$ можна записати у формі $A \to BD$. Тоді продукції $A \to BCD \mid BD$ охоплюють усі випадки для C: як $C = \varepsilon$, так і $C \neq \varepsilon$. Отже початкова граматика може бути записана у формі:

$$\begin{aligned} &A \to BCD \mid BD \\ &C \to F \mid G \\ &B \to w_1 \\ &D \to w_2 \\ &F \to w_3 \\ &G \to w_4 \end{aligned}$$

Отримана граматика не містить ε-продукцій і еквівалентна початковій.

Усунення ланцюгових правил

Граматику називають граматикою без циклів (ациклічною граматикою), якщо в ній неможливе виведення $A \stackrel{+}{\Rightarrow} A$, де $A \in N$. Щоб граматика була ациклічною, з неї необхідно видалити ланцюгові продукції – так називають продукції типу $A \to B$, де $A, B \in N$.

Для вилучення ланцюгових продукцій треба кожну ланцюгову продукцію $A \to B$ замінити на продукції $A \to \alpha$ за умови, що $B \to \alpha$ не є ланцюговою.

Наприклад, граматика

$$A \to BCD$$
$$C \to A \mid cd$$
$$D \to C$$
$$B \to b$$

містить ланцюгові продукції $C \to A$ та $D \to C$. Заміна в першій із них символа A на його визначення BCD (з неланцюгової продукції $A \to BCD$) приводить граматику до форми:

$$A \to BCD$$
$$C \to BCD \mid cd$$
$$D \to C$$
$$B \to b$$

Замінивши у $D \to C$ символ C на варіанти його визначення з неланцюгових продукцій $C \to BCD \mid cd$, знаходимо граматику, еквівалентну початковій:

$$A \to BCD$$
$$C \to BCD \mid cd$$
$$D \to BCD \mid cd$$
$$B \to b$$

Усунення марних символів

Символ X називають продуктивним, якщо з нього може бути виведене речення, тобто $X \stackrel{*}{\Rightarrow} w$, де $w \in T^*$. Термінали вважають продуктивними символами, що породжують речення за нуль кроків.

Символ X називають досяжним, якщо він може бути виведений зі стартового символа S, тобто $S \stackrel{*}{\Rightarrow} \alpha X \beta$, де $\alpha, \beta \in (T \cup N)^*$.

Символ X називають суттєвим (корисним), якщо він є і продуктивним, і досяжним. Інакше його називають несуттєвим (некорисним, марним).

Для видалення некорисних символів спочатку видаляють непродуктивні символи та продукції, що містять один чи кілька таких символів, після чого видаляють усі недосяжні символи.

Наприклад, продукції

$$A \to BC \mid a$$
$$B \to b$$

містять продуктивні символи a, b, S, B, та непродуктивний C. Тому продукція $A \to BC$ видаляється, і граматика приводиться до форми:

$$A \to a$$
$$B \to b$$

Видаливши недосяжний символ B, знаходимо граматику, еквівалентну початковій:

$$A \to a$$

Усунення лівої рекурсії

Ліворекурсивна граматика – така, в що в ній існує виведення $A \stackrel{+}{\Rightarrow} A\alpha$, де $\alpha \in (T \cup N)^*$, може бути приведена до еквівалентної форми без лівої рекурсії.

Для того, щоб нетермінал A був суттєвим символом, маючи ліворекурсивне визначення, необхідно, щоб існували продукції $A \to A\alpha$ та $A \to \beta$. Зауважимо, що n-разове застосування першої з наведених продукцій приведе до слова $A\alpha^n$, а наступне застосування другої продукції – до $\beta\alpha^n$.

Для усунення лівої рекурсії у продукції типу $A \to A\alpha \mid \beta$ вводять додатковий нетермінал і будують еквівалентні правила без лівої рекурсії у формі:

$$A \to \beta B$$
$$B \to \alpha B \mid \varepsilon$$

Факторизація граматик

Спільний префікс у різних визначеннях певного нетермінала призводить до ускладнення при розборі вхідного рядка. Для усунення такої проблеми виконують (ліву) факторизацію таких правил, яка зводиться до винесення за дужки спільного префікса кількох альтернатив продукції.

Так, якщо $A \to \alpha \mid \alpha\beta \mid \alpha\gamma$, то еквівалентною формою буде $A \to \alpha(\varepsilon \mid \beta \mid \gamma)$.

Нормальна форма Чомскі

Кажуть, що граматика $\Gamma = (N, T, P, S)$ знаходиться у нормальній формі Чомскі (НФЧ), якщо кожна її продукція має одну з таких форм:

$$A \to BC$$
$$A \to a$$

де $a \in T$, $A, B, C \in N$.

Кожна контекстно вільна граматика може бути приведена до нормальної форми Чомскі.

Щоб трансформувати граматику $\Gamma = (N, T, P, S)$ у граматику в нормальній формі Чомскі $\Gamma' = (N', T, P', S)$, можна скористатись таким алгоритмом:

1. усі правила у формі $A \to BC$, $A \to a$ перенести із P у P';
2. у кожній продукції із P у формі $A \to \alpha$, де $|\alpha| > 1, \alpha \in (T \cup N)^*$ кожний термінал $a \in T$ замінити нетерміналом Y_a і додати до P' правило $Y_a \to a$ для кожного такого a. Створені нетермінали Y_a додати до N';

3. тепер у кожній продукції $A \to X_1X_2 \dots X_n$ із P усі X_i є або нетерміналами з N, або нетерміналами Y_a з N', доданими на попередньому кроці.
   Кожну таку продукцію $A \to X_1X_2 \dots X_n$ представити у формі
   $A \to X_1Y_1$
   $Y_1 \to X_2Y_2$
   $\dots$
   $Y_{n-1} \to X_{n-1}X_n$
   Створені нетермінали Y_i додати до N';

В табл. 1.1 наведено приклад трансформації граматики $\Gamma = (N, T, P, S)$ з множиною продукцій P:

$$A \to aBCDDD$$
$$B \to DD$$
$$B \to b$$
$$C \to abccc$$

множиною нетерміналів $N = \{A, B, C, D\}$, терміналів $T = \{a, b, c\}$ та стартовим символом S до граматики у НФЧ $\Gamma' = (N', T, P', S)$.

У графах табл. 1.1 позначено: номер кроку алгоритму (Крок); продукцію, що трансформується (Продукція); власне трансформація – перепозначення або перенесення правил, що вже відповідають формі НФЧ до множини P', (Трансформація); уже приведені до НФЧ правила (P'); множина нетерміналів граматики Γ' на поточному кроці алгоритму (N').

Після виконання кроку 3 алгоритму для кожного правила граматики Γ маємо еквівалентну їй граматику у нормальній формі Чомскі $\Gamma' = (N', T, P', S)$ з множиною продукцій P':

$$A \to Y_aY_1$$
$$Y_1 \to BY_2$$
$$Y_2 \to CY_3$$
$$Y_3 \to DY_4$$
$$Y_4 \to DD$$
$$Y_a \to a$$
$$B \to DD$$
$$B \to b$$
$$C \to Y_aY_bY_cY_cY_c$$
$$Y_5 \to Y_bY_6$$
$$Y_6 \to Y_cY_7$$
$$Y_7 \to Y_cY_c$$
$$Y_b \to b$$
$$Y_c \to c$$
$$D \to AB$$

множиною нетерміналів $N' = \{A, B, C, D, Y_a, Y_b, Y_c, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, Y_7\}$, множиною терміналів $T = \{a, b, c\}$ і стартовим символом S.

Таблиця 1.1. Приведення граматики до нормальної форми Чомскі

Крок	Продукція	Трансформація	P'	N'
		P : $A \to aBCDDD$ $B \to DD$ $B \to b$ $C \to abccc$ $D \to AB$		$\{A, B, C, D\}$
1	 ✓ ✓ ✓	$A \to aBCDDD$ $C \to abccc$ 	 $B \to DD$ $B \to b$ $D \to AB$	
2	✓ ✓	$A \to Y_aBCDDD$ $C \to Y_aY_bY_cY_cY_c$	 $Y_a \to a$ $B \to DD$ $B \to b$ $Y_b \to b$ $Y_c \to c$ $D \to AB$	$\{A, B, C, D,$ $Y_a, Y_b, Y_c\}$
3	✓ ✓	$A \to Y_aBCDDD$ $C \to Y_aY_bY_cY_cY_c$	$A \to Y_aY_1$ $Y_1 \to BY_2$ $Y_2 \to CY_3$ $Y_3 \to DY_4$ $Y_4 \to DD$ $Y_a \to a$ $B \to DD$ $B \to b$ $Y_b \to b$ $Y_c \to c$ $D \to AB$	$\{A, B, C, D,$ $Y_a, Y_b, Y_c,$ $Y_1, Y_2, Y_3, Y_4\}$
3	 ✓	 $C \to Y_aY_bY_cY_cY_c$	$A \to Y_aY_1$ $Y_1 \to BY_2$ $Y_2 \to CY_3$ $Y_3 \to DY_4$ $Y_4 \to DD$ $Y_a \to a$ $B \to DD$ $B \to b$ $C \to Y_aY_5$ $Y_5 \to Y_bY_6$ $Y_6 \to Y_cY_7$ $Y_7 \to Y_cY_c$ $Y_b \to b$ $Y_c \to c$ $D \to AB$	$\{A, B, C, D,$ $Y_a, Y_b, Y_c,$ $Y_1, Y_2, Y_3, Y_4,$ $Y_5, Y_6, Y_7\}$

1.4. Розпізнавачі

Для визначення мов використовують розпізнавачі – абстрактні пристрої для моделювання обчислювального процесу, зокрема для визначення множини слів певної мови.

В загальному випадку, як це видно з рис. 1.15, розпізнавач складається з вхідної стрічки, яку вважають розділеною на комірки, у кожній з яких може міститись єдиний символ вхідного алфавіту. Інколи за останнім символом слова та/чи перед першим записують спеціальні обмежувальні символи, що не належать вхідному алфавіту. В кожний момент часу одну з комірок оглядає вхідна головка. В загальному випадку вона може прочитати символ, записаний у комірці, або стерти наявний, або записати[1] туди інший. У двох останніх випадках символ, який був у комірці раніше, втрачається. Головка може також перейти до комірки ліворуч чи праворуч від поточної, або залишитись на місці.

Якщо головка може рухатись тільки в один бік, то такий розпізнавач називають однобічним.

Вважається, що керуючий пристрій має скінченну пам'ять. І хоча ця пам'ять може бути наскільки завгодно великою, все ж вона обмежена певним, хай і великим, значенням. Зазвичай пам'ять керуючого пристрою представляють як певну множину станів. Один із станів вважається стартовим, один або кілька складають множину заключних, або фінальних станів.

Додаткова робоча пам'ять, якщо вона використовується, може бути обмеженою або ні. Алфавіт робочої пам'яті може відрізнятися від вхідного алфавіту.

Вважається, що розпізнавач працює у дискретному часі, тобто час розділений на окремі відрізки – такти, на кожному з яких здійснюються певні кроки: вхідна головка переміщується до наступної комірки, або не переміщується; з робочої пам'яті зчитується, або не зчитується, або в неї записується певна інформація; з поточного стану керуюючий пристрій переходить або не переходить до наступного.

Конфігурацією розпізнавача на певному такті називають вміст вхідної стрічки, разом з позицією головки, поточний стан керуючого пристрою та стан робочої пам'яті.

На початку роботи керуючий пристрій встановлюється у стартовий стан – головка оглядає найперший символ вхідної стрічки і, можливо, робоча пам'ять містить необхідну для роботи інформацію. Таку конфігурацію називають початковою.

Заключною називають конфігурацію, коли головка оглядає комірку вхідної стрічки, наступну за тією, що містить останній символ вхідного слова, а керуючий пристрій знаходиться в одному із заключних станів, і виконані вимоги до вмісту робочої пам'яті, якщо вони є.

З довільного поточного стану розпізнавач переходить у наступний стан відповідно до функції переходів, значення якої залежить від прочитаного символу, поточного стану та стану робочої пам'яті, а також здійснює необхідні дії, визначені конструкцією розпізнавача (іншими функціями).

[1] У власне розпізнавачах використовується рідко.

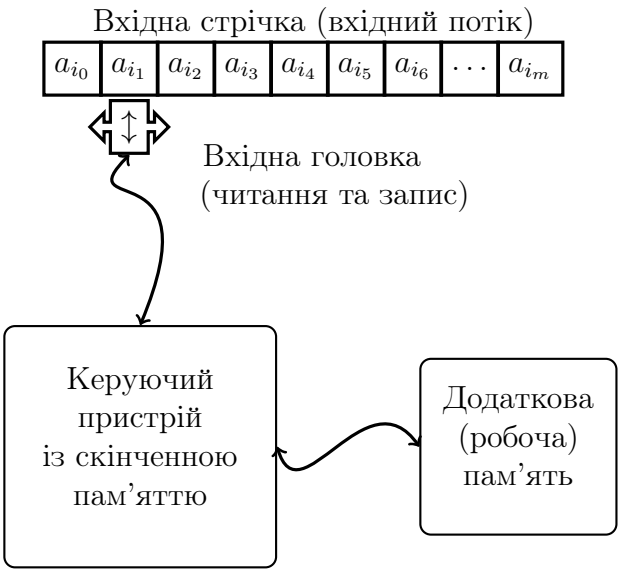

Рис. 1.15. Загальна схема розпізнавача

Кажуть, що розпізнавач розпізнає, чи допускає, вхідне слово, якщо при його обробці розпізнавач переходить від початкової до заключної конфігурації. Мова, яку визначає розпізнавач, це множина слів, які допускає цей розпізнавач.

1.4.1. Скінченні автомати

Найпростішими з розпізнавачів є скінченні автомати. Скінченні автомати здатні розпізнавати автоматні мови, див. класифікацію на стор. 24, що й обумовило назву цього класу мов.

Схема скінченного автомата представлена на рис. 1.16, з якої можна бачити, що вхідна головка може тільки читати символи вхідної стрічки і не може їх записувати. Головка у кінці кожного такту переміщується до наступної справа комірки, отже скінченний автомат – однобічний.

Керуючий пристрій має n, скінченну кількість, станів. Стани, за необхідності, нумеруються або позначаються іншим чином. Початковий стан часто позначають як 0, або як q_0. Множину заключних станів задають їх переліком.

Функція переходів[1], її часто позначають символом δ, визначає наступний стан в залежності від поточного стану та прочитаного символу. Автомат називають повним, якщо функція δ визначена для усіх символів вхідного алфавіту у кожному поточному стані, інакше автомат називають частковим (тобто, якщо є хоч один такий стан, для якого за певним вхідним символом перехід не визначений).

Додаткової (робочої) пам'яті скінченні автомати не використовують.

Представляють скінченні автомати або у формальному символьному вигляді, або графічно у формі орієнтованого розміченого графа (графа переходів),

[1] У загальному випадку – відображення.

або у формі таблиці переходів.

Серед скінченних автоматів розрізняють детерміновані (ДСА) та недетерміновані (НСА), які, попри відмінності, мають однакову виразну здатність, тобто здатні розпізнавати одну й ту ж множину мов.

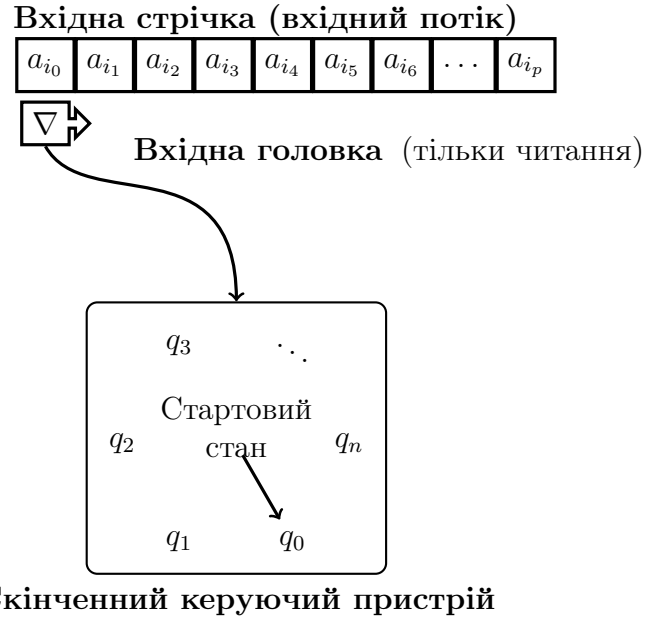

Рис. 1.16. Схема скінченного автомата

Детерміновані скінченні автомати (ДСА)

Детермінваним скінченним автоматом називають п'ятірку $M = (Q, \Sigma, \delta, q_0, F)$, де Q – скінченна множина станів,

Σ – скінченний вхідний алфавіт,

δ – функція переходів $\delta : Q \times \Sigma \to Q$,

q_0 – початковий (стартовий) стан,

F – множина заключних (кінцевих) станів.

Детермінованість (визначеність) автомата полягає у тому, що:

- на кожному такті вхідна головка обов'язково зчитує один вхідний символ (відповідно переміщується праворуч до наступної вхідної комірки);
- $\delta(q_{j_m}, a_{i_k})$ – є функцією, тобто з кожного поточного стану q_{j_m} за одним і тим же вхідним символом a_{i_k} відображення δ повертає не більше одного значення (тобто з будь-якого стану за одним і тим же вхідним символом можна перейти тільки в один наступний стан.

Наприклад, детермінований скінченний автомат M_1, мову якого складають слова над алфавітом $\Sigma = \{a, b\}$, в яких символи алфавіту по черзі змінюють

один одного (тобто $L(\Sigma) = \{a, b, ab, ba, aba, bab, \dots\}$), може бути представлений так

$$M_1 = (\{q_0, q_1, q_2\}, \{a, b\}, \delta, q_0, \{q_1, q_2\})$$

де $\delta = \{\delta(q_0, a) = q_1, \delta(q_0, b) = q_2, \delta(q_1, b) = q_2, \delta(q_2, a) = q_1\}$.

При графічному представленні, див. рис. 1.17 початковий стан позначають стрілкою, заключні стани – подвійним колом, переходи – поміченими стрілками.

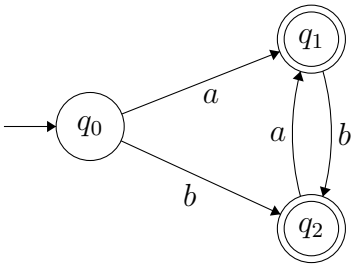

Рис. 1.17. Детермінований скінченний автомат M_1

Той же ДСА M_1 можна подати у табличній формі, див. табл. 1.2:

Таблиця 1.2. Таблиця переходів ДСА M_1

Поточний стан	Алфавіт Σ	
	a	b
q_0	q_1	q_2
q_1		q_2
q_2	q_1	

З усіх наведених представлень видно, особливо наочно – з табличного, що автомат M_1 є частковим, оскільки переходи зі стану q_1 за символом a та зі стану q_2 за символом b не визначені. Відсутність переходу (порожня комірка у табличному представленні) означає, що вхідне слово, яке потребує такого переходу буде відхилене (автомат не допускає такого слова; таке слово не належить мові, яку розпізнає автомат M_1).

Недетерміновані скінченні автомати (НСА)

Недетермінований скінченний автомат має такі особливості:

- на кожному такті вхідна головка може прочитати порожній символ ε, і тоді вона не переміщується до наступної вхідної комірки, адже ще не був прочитаний символ з поточної комірки;
- $\delta(q_{j_m}, a_{i_k})$ – не обов'язково є функцією, тобто для кожного поточного стану q_{j_m} за одним і тим же вхідним символом a_{i_k} відображення δ повертає множину наступних станів (зокрема й порожню для часткового автомата).

Формально недетермінованим скінченним автоматом називають п'ятірку

$$M = (Q, \Sigma, \delta, q_0, F),$$

де Q – скінченна множина станів,
Σ – скінченний вхідний алфавіт,
δ – "функція" переходів (відображення) $\delta : Q \times (\Sigma \cup \varepsilon) \to \{Q_{i_s}\}$,
q_0 – початковий (стартовий) стан,
F – множина заключних (кінцевих) станів.

Побудуємо недетермінований скінченний автомат M_2 (еквівалентний ДСА M_1), який визначає мову зі словами над алфавітом $\Sigma = \{a, b\}$, де символи алфавіту по черзі змінюють один одного (тобто $L(\Sigma) = \{a, b, ab, ba, aba, bab, \dots\}$), рис. 1.18:

$$M_2 = (\{q_0, q_1, q_2, q_3, q_4\}, \{a, b\}, \delta, q_0, \{q_3, q_4\}),$$

де $\delta = \{\delta(q_0, \varepsilon) = q_0, \delta(q_0, \varepsilon) = q_1, \delta(q_0, \varepsilon) = q_2, \delta(q_1, a) = q_3, \delta(q_3, b) = q_4, \delta(q_2, b) = q_4, \delta(q_4, a) = q_5, \delta(q_4, a) = q_3, \delta(q_5, \varepsilon) = q_3\}$.

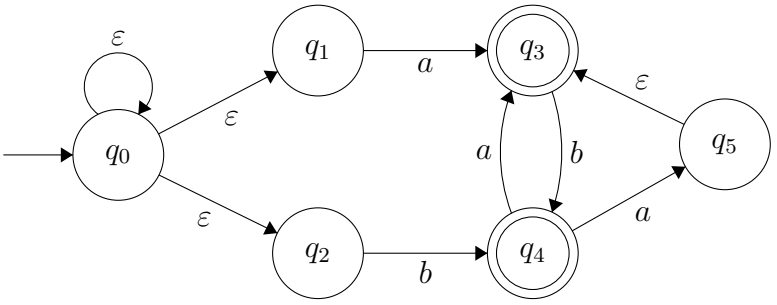

Рис. 1.18. Недетермінований скінченний автомат M_2

Недетермінований скінченний автомат M_2 може бути представлений у табличній формі, див. табл. 1.3.

Таблиця 1.3. Таблиця переходів НСА M_2

Поточний стан	ε	a	b
q_0	$\{q_0, q_1, q_2\}$		
q_1		q_3	
q_2			q_4
q_3			q_4
q_4		$\{q_2, q_5\}$	
q_5	q_3		

Скінченні автомати, праволінійні граматики та регулярні вирази

Еквівалентність регулярних та автоматних мов була проілюстрована прикладом побудови розміченого графа регулярного виразу у розд. 1.2.

Тепер розглянемо побудову праволінійної граматики за НСА. Це можна зробити, дотримуючись таких правил:

1. кожному стану автомата поставити у відповідність нетермінал;
2. переходу зі стану A у стан B за символом a поставити у відповідність продукцію $A \to aB$;
3. кожному заключному стану A поставити у відповідність продукцію $A \to \varepsilon$;
4. кожному ε-переходу з A до B поставити у відповідність продукцію $A \to B$.

Приклад. Побудуємо праволінійну граматику, еквівалентну НСА M_2, див. табл. 1.3.

За правилом 1 поставимо нетермінали S, A, B, C, D, E у відповідність стнанам $q_0, q_1, q_2, q_3, q_4, q_5$ відповідно.

За правилом 2 запишемо продукції, що відповідають переходам за символами a та b:

$$A \to aC$$
$$B \to bD$$
$$C \to bD$$
$$D \to aC$$
$$D \to aE$$

За правилом 3 запишемо продукції,, що відповідають заключним станам:

$$C \to \varepsilon$$
$$D \to \varepsilon$$

За правилом 4 запишемо продукції, що відповідають ε-переходам:

$$S \to S$$
$$S \to A$$
$$S \to B$$
$$E \to C$$

Або, більш компактно, та видаливши тавтологічне правило $S \to S$:

$$S \to A \mid B$$
$$A \to aC$$
$$B \to bD$$
$$C \to bD \mid \varepsilon$$
$$D \to aC \mid aE \mid \varepsilon$$
$$E \to C$$

1.4.2. Автомати з магазинною пам'яттю

Автомат з магазинною пам'яттю (МП-автомат) – це однобічний розпізнавач, який використовує додаткову (робочу) пам'ять – стек (магазин).

Стек, як відомо, у кожний момент часу надає доступ тільки до вершини. На кожному такті МП-автомат зчитує вхідний символ та переходить до огляду

наступної комірки вхідної стрічки, знімає символ з вершини стека, переходить до наступного стану, та, можливо, кладе якийсь символ на стек, див. Рис. 1.19. МП-автомат не переходить до огляду наступної комірки вхідної стрічки, якщо був прочитаний порожній рядок ε. Вміст стека та елемент у вершині стека лишаються незмінними, якщо був знятий зі стека або покладений на стек порожній рядок ε.

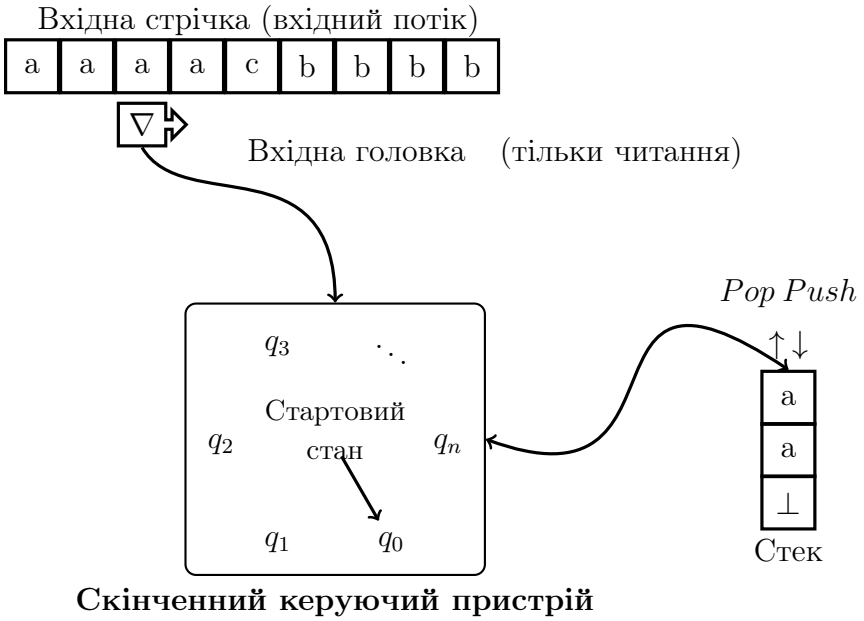

Рис. 1.19. Схема магазинного автомата

Формально автоматом з магазинною пам'яттю називають сімку

$$M = (Q, \Sigma, \Gamma, \delta, q_0, T_0, F),$$

де Q – скінченна множина станів,
Σ – скінченний вхідний алфавіт,
Γ – магазинний алфавіт,
δ – функція переходів $\delta : Q \times (\Sigma \cup \varepsilon) \times (\Gamma \cup \varepsilon) \to Q \times (\Gamma \cup \varepsilon)$,
q_0 – початковий (стартовий) стан,
T_0 – маркер порожнього стека, на рис. 1.19 позначений символом $\perp$,
F – множина заключних станів.

Для опису стану МП-автомата під час розбору слова використовують конфігуції. Конфігурація $(q_i, aw, X\alpha)$ на певному такті відображає той факт, що МП-автомат знаходиться у стані q_i, вхідна головка оглядає комірку з символом a (необроблений суфікс слова – aw), на вершині стека – символ X (вміст стека – $X\alpha$).

Якщо поточною конфігурацією є $(q_i, aw, X\alpha)$, а функція переходів від таких аргументів $\delta(q_i, a, X)$ містить значення (q_j, Y) і МП-автомат здійснить такий перехід, то наступною конфігурацією буде $(q_j, w, Y\alpha)$. Тоді кажуть, що за конфі-

гурацією $(q_i, aw, X\alpha)$ безпосередньо слідує конфігурація $(q_j, w, Y\alpha)$, і записують $(q_i, aw, X\alpha) \vdash (q_j, w, Y\alpha)$.

Якщо за нуль чи більше переходів МП-автомат з конфігурації (q_i, w_1, α_1) переходить до (q_j, w_2, α_2), то кажуть, що (q_j, w_2, α_2) слідує за (q_i, w_1, α_1) і записують $(q_i, w_1, \alpha_1) \overset{*}{\vdash} (q_j, w_2, \alpha_2)$. Послідовність переходів (як і послідовність допустимих конфігурацій) МП-автомата називають обчислювальним шляхом.

Початкова конфігурація може бути представлена у таких формах:

1. (q_0, w, ε) – стартовий внутрішній стан q_0, вхідне слово w, стек порожній. Передбачається, що маркер порожнього стека T_0 буде покладений на стек на першому переході та знятий зі стека при переході до заключної конфігурації;

2. (q_0, w, T_0) – стартовий внутрішній стан q_0, вхідне слово w, але вважається, що порожній стек завжди містить маркер дна стека.

Заключною вважають одну з двох конфігурацій:

1. заключний стан, вхідне слово прочитане, стек порожній:

   - $(q_j, \varepsilon, \varepsilon), q_j \in F$;
   - $(q_j, \varepsilon, T_0), q_j \in F$, якщо вважається що порожній стек завжди містить маркер дна стека;

2. $(q_j, \varepsilon, \alpha)$, де $q_j \in F, \alpha \in \Gamma^*$ – заключний стан, вхідне слово прочитане, стек містить довільне слово над алфавітом стекових символів.

МП-автомат M розпізнає слово w, якщо існує обчислювальний шлях з початкової конфігурації до заключної. Відповідно, розрізняють допустимість слова w за:

1. заключним станом і порожнім стеком $(q_0, w, \varepsilon) \overset{*}{\vdash} (q_j, \varepsilon, \varepsilon)$, де $q_j \in F$,
   або $(q_0, w, T_0) \overset{*}{\vdash} (q_j, \varepsilon, T_0)$, де $q_j \in F$, якщо порожній стек завжди містить маркер дна стека;

2. заключним станом $(q_0, w, \varepsilon) \overset{*}{\vdash} (q_j, \varepsilon, \alpha)$, де $q_j \in F, \alpha \in \Gamma^*$, або, відповідно, $(q_0, w, T_0) \overset{*}{\vdash} (q_j, \varepsilon, \alpha)$.

Приклад. Побудуємо МП-автомат M_3 для мови $(L = \{a^n b^n \mid n \geq 1\})$. Для цього спочатку покладемо маркек T_0 на стек, після чого кожну літеру a вхідного слова покладемо на стек, залишаючись у стані q_1. За вхідним символом b перейдемо у стан q_2, залишаючись у ньому, поки на вході – літери b вхідного слова. Кожне зчитування символу b супроводжується зняттям зі стека символу a. Отже, у вхідному слові можна прочитати рівно стільки символів b, скільки символів a було записано у стек. Після зчитування останнього символу b вхідна

стрічка порожня, а стек містить тільки маркер дна стека. Якщо це так, то залишається зняти зі стека маркер T_0 і перейти до заключного стану q_3 з порожнім стеком, див. рис. 1.20.

$$M_3 = (\{q_0, q_1, q_2, q_3\}, \{a, b\}, \{a, b, T_0\}, \delta, q_0, T_0, \{q_3\}),$$

де функція переходів
$\delta(q_0, \varepsilon, \varepsilon) = (q_1, T_0)$
$\delta(q_1, a, \varepsilon) = (q_1, a)$
$\delta(q_1, b, a) = (q_2, \varepsilon)$
$\delta(q_2, b, a) = (q_2, \varepsilon)$
$\delta(q_2, \varepsilon, T_0) = (q_3, \varepsilon)$

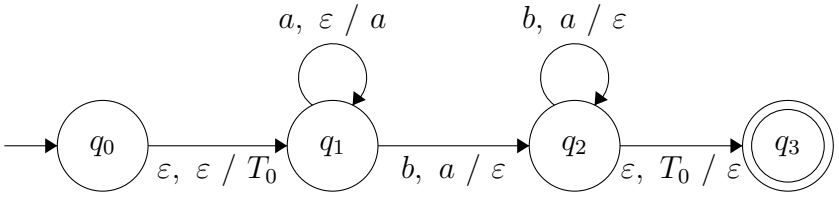

Рис. 1.20. Діаграма переходів МП-автомата M_3

Приклад. Побудуємо МП-автомат M_4, еквівалентний автомату M_3. Побудова повністю аналогічна попередньому прикладу, проте тут не використовується маркер порожнього стека T_0 (можна вважати, що прийнято $T_0 = \varepsilon$), див. рис. 1.21.

$$M_3 = (\{q_0, q_1, q_2\}, \{a, b\}, \{a, b, T_0\}, \delta, q_0, T_0, \{q_2\}),$$

де
$\delta(q_0, a, \varepsilon) = (q_1, \varepsilon)$
$\delta(q_1, a, \varepsilon) = (q_1, a)$
$\delta(q_1, b, a) = (q_2, \varepsilon)$
$\delta(q_2, b, a) = (q_2, \varepsilon)$

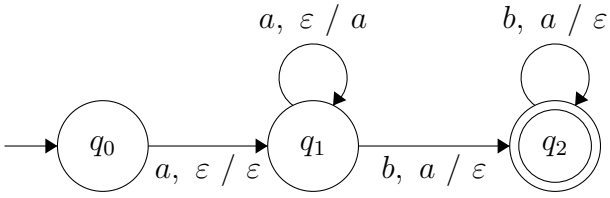

Рис. 1.21. Діаграма переходів МП-автомата M_4

Приклад. Побудуємо МП-автомат M_5, який розпізнає мову

$$(L = \{a^{2m} b^{3n} c^k \mid m, n, k \geq 1, 2m + 3n = k\}).$$

Префікс кожного слова цієї мови, який містить кілька пар літер a та кілька трійок літер b, має довжину, рівну довжині суфікса, який складається тільки з літер c:

$$L(M_5) = \{aabbbccccc, aaaabbbccccccc, aabbbbbbccccccccc, \dots\},$$

або, те саме,

$$L(M_5) = \{a^2b^3c^5, a^4b^3c^7, a^2b^6c^8, \dots\}.$$

Маркер T_0 використовувати не будемо, визначатимемо допустимість слова по заключному стану та порожньому стеку.

Читатимемо попарно літери a переходячи зі стану q_0 у q_1 та з q_1 – у q_0, при кожному читанні вхідного a записуватимемо його у стек. Коли символи a вхідного слова будуть прочитані, перейдемо до стану q_2, впевнившись, що у стеку є літери a, – для цього знімемо і знову покладемо на стек літеру a.

Переходи $q_2 \to q_3 \to q_4 \to q_2$ забезпечують читання вхідного символа b трійками та їх запис у стек.

Далі з вхідної стрічки зчитуватимуться символи c, при кожному зчитуванні зі стека зніматимуться наявні там літери: спочатку b, потім a (за першою c – перехід у заключний стан q_5, решта – залишаючись у заключному стані q_5), див. рис. 1.22.

Розглянемо два відмінних обчислювальних шляхи для слова

$$aaaabbbccccc = a^4b^3c^7.$$

Перший шлях не розпізнає слова:
$(q_0, aaaab^3c^7, \varepsilon) \vdash (q_1, aaab^3c^7, a) \vdash (q_0, aab^3c^7, aa) \vdash (q_2, aab^3c^7, aa)$,
оскільки здійснюється можливий прехід $\delta(q_0, \varepsilon, a) = (q_2, a)$ без зчитування літери a, а зі стану q_2 немає переходу із зчитуванням літер a.

Другий шлях – розпізнає:
$(q_0, a^4b^3c^7, \varepsilon) \vdash (q_1, a^3b^3c^7, a) \vdash (q_0, a^2b^3c^7, a^2) \vdash (q_1, ab^3c^7, a^3) \vdash (q_0, b^3c^7, a^4) \vdash (q_2, b^3c^7, a^4) \vdash (q_3, b^2c^7, ba^4) \vdash (q_4, bc^7, b^2a^4) \vdash (q_2, c^7, b^3a^4) \vdash (q_5, c^6, b^2a^4) \vdash (q_5, c^5, ba^4) \vdash (q_5, c^4, a^4) \vdash (q_5, c^3, a^3) \vdash (q_5, c^2, a^2) \vdash (q_5, c, a) \vdash (q_5, \varepsilon, \varepsilon)$.

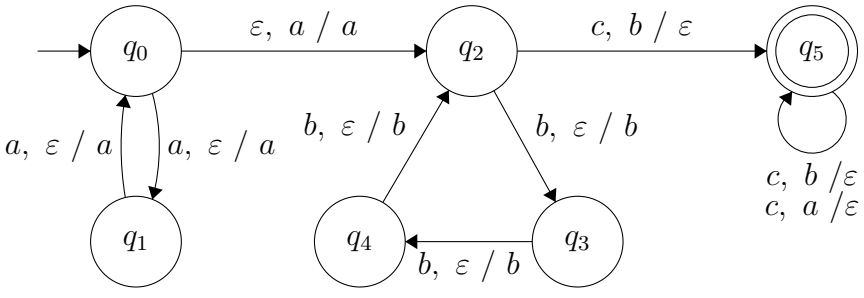

Рис. 1.22. Діаграма переходів МП-автомата M_5

Детермінований магазинний автомат – це МП-автомат, у якому:

1. функція переходів $\delta(q_i, a, X)$, має не більше одного значення для кожного з $q_i \in Q, a \in (\Sigma \cup \varepsilon), X \in \Gamma$;

2. можливий тільки один із двох переходів $\delta(q_i, a, X)$ та $\delta(q_i, \varepsilon, X)$ для довільних $q_i \in Q, X \in \Gamma$.

МП-автомат M_5 задовольняє першій, але не другій з наведених вимог, оскільки зі стану q_0 з символом a на вершині стека можливі переходи як за вхідним символом ε, так і за іншим (a) вхідним символом.

Детермінований магазинний автомат (ДМП-автомат) M_6, еквівалентний автомату M_5, наведений на рис. 1.23.

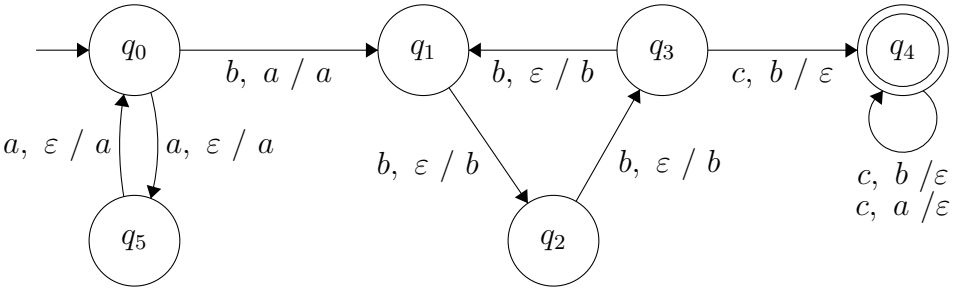

Рис. 1.23. Діаграма переходів ДМП-автомата M_6

МП-автомати та контекстно вільні граматики

Контекстно вільна граматика $\Gamma = (N, \Sigma, P, S)$ може бути трансформована у МП-автомат $M = (\{q\}, \Sigma, N \cup \Sigma, \delta, q, S, \varnothing)$ з допуском по порожньому стеку, якщо функція переходів δ побудована за правилами:

1. для кожної продукції із P виду $A \to \alpha$, де $A \in N, \alpha \in (N \cup T)^*$ функція $\delta(q, \varepsilon, A)$ містить значення (q, α);

2. для кожного термінала $a \in \Sigma$ функція $\delta(q, a, a)$ містить значення (q, ε).

Отже, побудований МП-автомат має єдиний внутрішній стан q, маркером дна стека приймається стартовий символ граматики, а правила переходу зводяться або до заміни на вершині стека нетермінала його визначенням – правою частиною відповідної продукції (правило 1), або до зчитування термінального символа вхідного слова та такого ж символа з вершини стека (правило 2). Заключною конфігурацією є $(q, \varepsilon, \varepsilon)$.

Розглянемо трансформацію контекстно вільної граматики
$\Gamma = (\{A, S\}, \{a, b\}, P, S)$, що визначає мову $L(\Gamma) = \{a^n c b^n \mid n \geq 1\}$:
тут P – множина продукцій:

$$S \to aAb$$
$$A \to aAb$$
$$A \to c$$

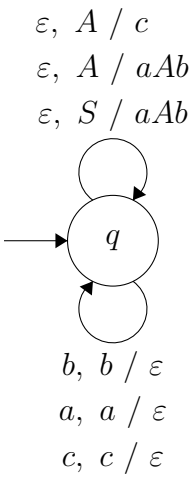

Рис. 1.24. Діаграма переходів ДМП-автомата M_7

Тоді еквівалентний граматиці $L(\Gamma)$ МП-автомат визначається як:

$$M_7 = (\{q\}, \{a, b\}, \{a, b, S, A\}, \delta, q, S, \varnothing),$$

де функція переходів:
$\delta(q, \varepsilon, S) = (q, aAb)$, за правилом 1;
$\delta(q, \varepsilon, A) = \{(q, aAb), (q, c)\}$, за правилом 1;
$\delta(q, a, a) = (q, \varepsilon)$, за правилом 2;
$\delta(q, b, b) = (q, \varepsilon)$, за правилом 2;
$\delta(q, c, c) = (q, \varepsilon)$, за правилом 2.

Граф переходів МП-автомата M_7 показаний на рис. 1.24.

1.4.3. Лінійно обмежені автомати та машина Тьюринга

Контекстно залежні мови та мови без обмежень можуть бути розпізнані лінійно обмеженими автоматами та машинами Тьюринга відповідно.

Машина Тьюринга (МТ), як універсальна модель організації обчислень, може бути використана як розпізнавач. Усе сказане на стор. 31 – 32 про загальну схему розпізнавача, див. рис. 1.15, повністю стосується МТ з тим уточненням, що її робоча пам'ять – це ще одна стрічка, яку оглядає ще одна головка, здатна читати символи, стирати чи записувати їх на стрічку, та рухатись ліворуч або праворуч від поточної комірки, або залишатись на місці. Стрічка робочої пам'яті вважається нескінченною справа і на початку розпізнавання – порожня. Разом з тим для МТ, як розпізнавача, достатньо, щоб вхідна головка могла читати та переходити тільки в один бік (або не переходити, якщо прочитаний порожній символ ε), див. рис. 1.25.

Кажуть, що МТ розпізнала (допустила) слово w, якщо у процесі його читання, використовуючи робочу пам'ять для довільних маніпуляцій з проміжною інформацією, МТ зможе перейти у заключну конфігурацію (вхідне слово прочитане, внутрішній стан – заключний). Вміст робочої пам'яті при цьому не має значення.

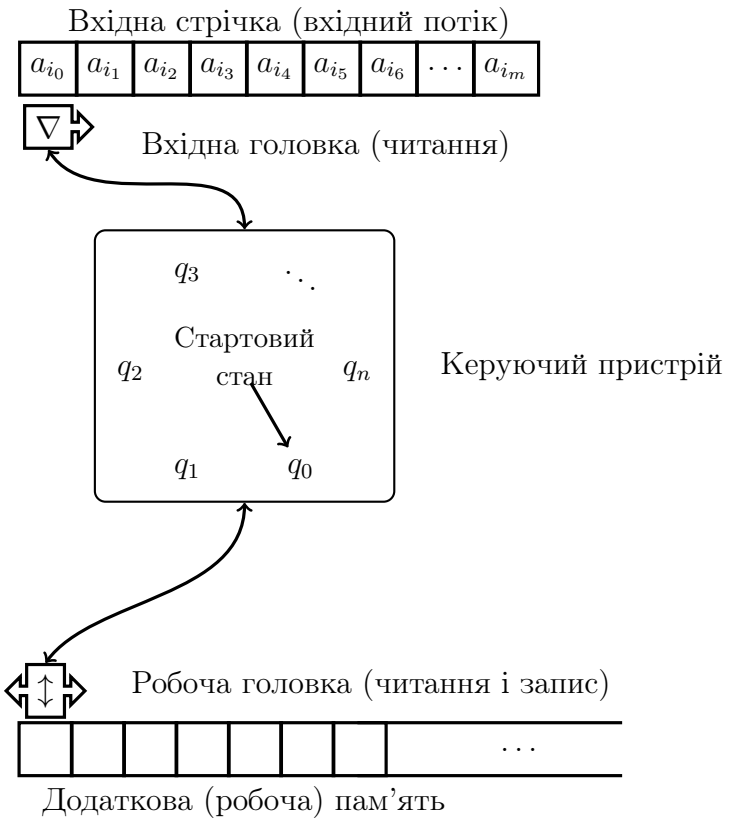

Рис. 1.25. Машина Тьюринга як розпізнавач

Машина Тьюринга здатна розпізнавати генеративні мови без обмежень (тип 0 за Чомскі), а відтак – і контекстно залежні, і контекстно вільні, і регулярні (автоматні) мови.

Лінійно обмеженим автоматом називають МТ, у якої довжина стрічки для проміжної інформації має скінченну довжину, що не перевищує $C * |w|$, де w – вхідне слово, C – константа, можливо велика. Іншими словами – це МТ з обмеженим обсягом робочої пам'яті.

Оскільки лінійно обмежені автомати здатні розпізнавати контекстно залежні мови (тип 1 за Чомскі), то, відтак, – і контекстно вільні, і регулярні (автоматні) мови.

1.5. Дидактичний інструментарій: JFLAP

Розробка формальної мови потребує спеціалізованих інструментів для перевірки її властивостей, представлення, трансформації тощо.

Для підтримки вивчення формальних мов та автоматів у вишах був розроблений пакет JFLAP (Java Formal Language and Automata Package), який надає низку зручних інструментів для створення, візуалізації, трансформації та конвертування автоматів, граматик та регулярних виразів [5, 6].

На рис. 1.26 можна бачити головне вікно програми та меню налаштування,

тут показано вибір способу позначення порожнього рядка (символу).

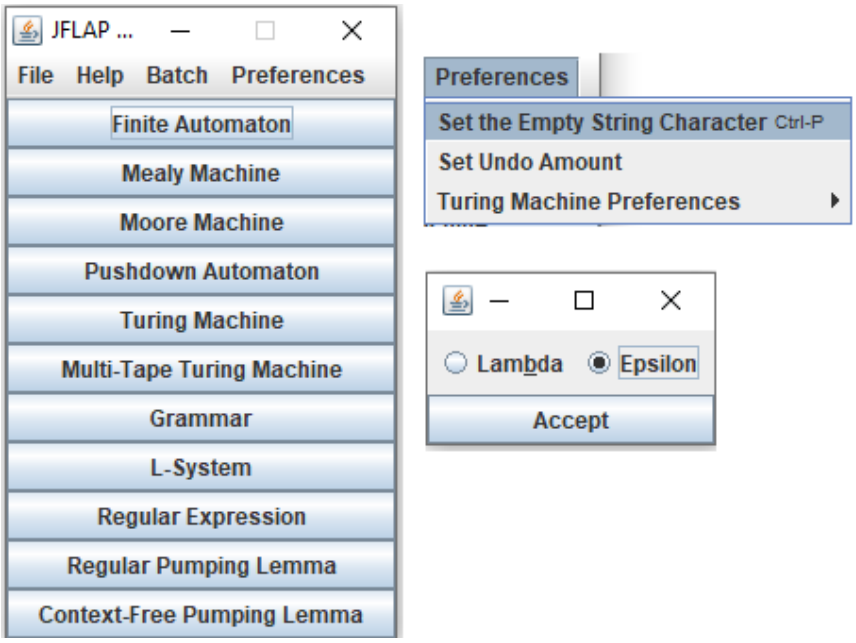

Рис. 1.26. Головне меню і налаштування

Натискання на кнопки головного вікна дозволяє викликати інструменти, зокрема, для роботи з скінченними автоматами (Finite Automata) – як детермінованими, так і недетермінованими, магазинними автоматами (Pushdown Automata), різного типу граматиками (Grammar), регулярними виразами (Regular Expression) тощо.

При роботі з автоматами довільного типу у вікні доступна панель інструментів, на рис. 1.27 виділена помаранчевим прямокутником: `(A)ttribute Editor`, `(S)tate Creator`, `(T)ransition Creator`, `(D)eleter` та дві кнопки `(U)ndoer` і `Undoe(r)` з очевидними за назвою функціями. Виділені у дужках символи є гарячими клавішами з тією умовою, що всі вони у нижньому регістрі. Тобто, натискання на кнопку клавіатури з літерою `s` активізує `(S)tate Creator`, а кнопки `t` — `(T)ransition Creator`.

1.5.1. Скінченні автомати

За допомогою інструмента `Attribute Editor` можна, по лівому кліку мишки, вибрати елемент автомата, а по правому – встановити його атрибути. Так, стан з іменем q_0), як це видно з рис. 1.27, може бути оголошений стартовим та/або заключним, можна встановити нове ім'я, прикріпити мітку тощо.

Інструмент `Deleter` видаляє довільний елемент автомата, а `State Creator`, по лівому кліку, створює новий стан. Перехід створюється, інструментом `Transition Creator`. Для цього треба перетягти мишкою лінію від одного стану до іншого та ввести символ, по якому може бути здійснений перехід. Якщо символ не ввести, то символ буде вважатись порожнім рядком ε (або λ).

Меню `Input` надає можливість симулювати роботу скінченного автомата на введеному рядку (рядках). Меню `Convert` надає можливість трансформації ав-

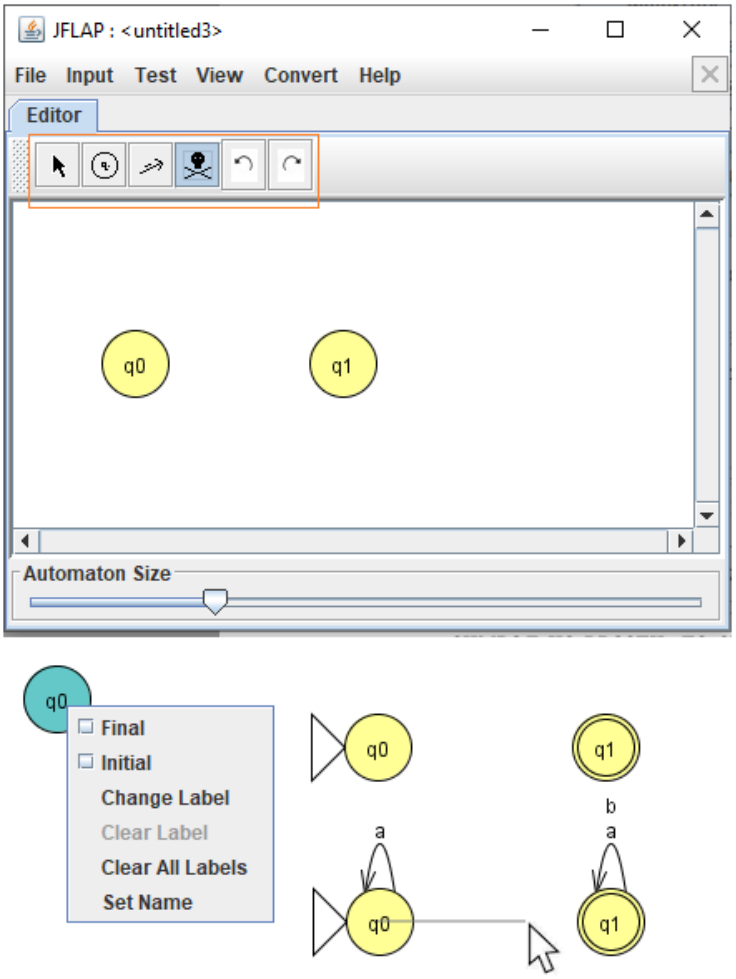

Рис. 1.27. Побудова скінченного автомата

томатів, їх конвертації та додавання стану-пастки. Обидва меню показані на рис. 1.28.

1.5.2. Магазинні автомати

Інструменти для роботи з магазинними автоматами відрізняються від вже розглянутих для скінченних автоматів тільки тим, що при створенні переходу інструментом `Transition Creator` слід ввести три символи: а) символ, що читається із вхідного потоку (`read`), б) символ, який має бути знятий з вершини стека (`pop`) і в) символ, який має бути покладений на вершину стека (`push`). Так, для переходу $q_0 \to q_1$ на рис. 1.29 це, відповідно, символи c, a і ε.

Магазинний автомат може бути побудований для одно- або багатосимвольного зчитування вхідного потоку, а симуляція роботи передбачає завершення роботи за заключним станом з вимогою порожнього стека, або без такої вимоги.

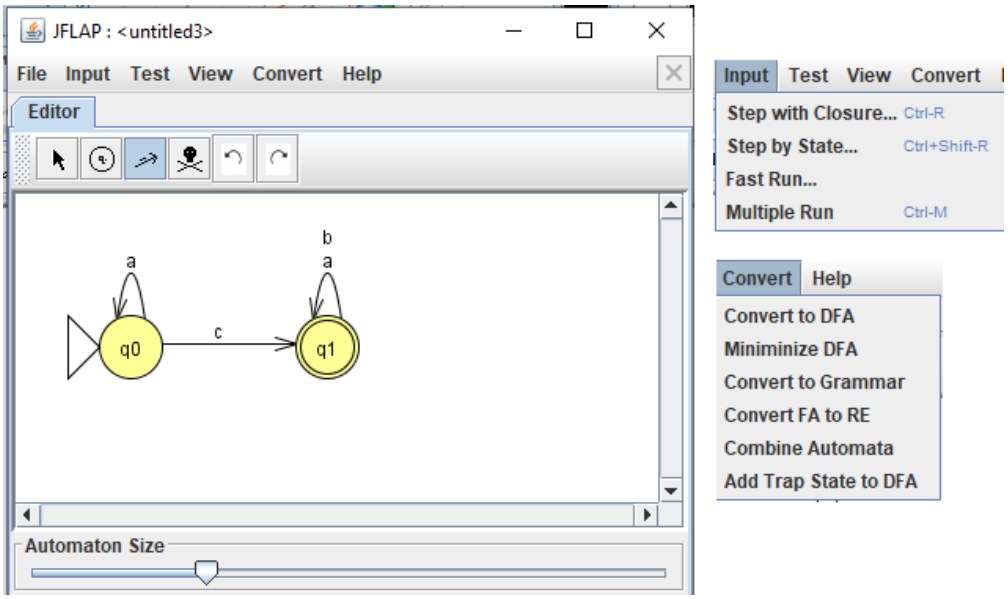

Рис. 1.28. Скінченний автомат, меню симуляції введення та конвертації

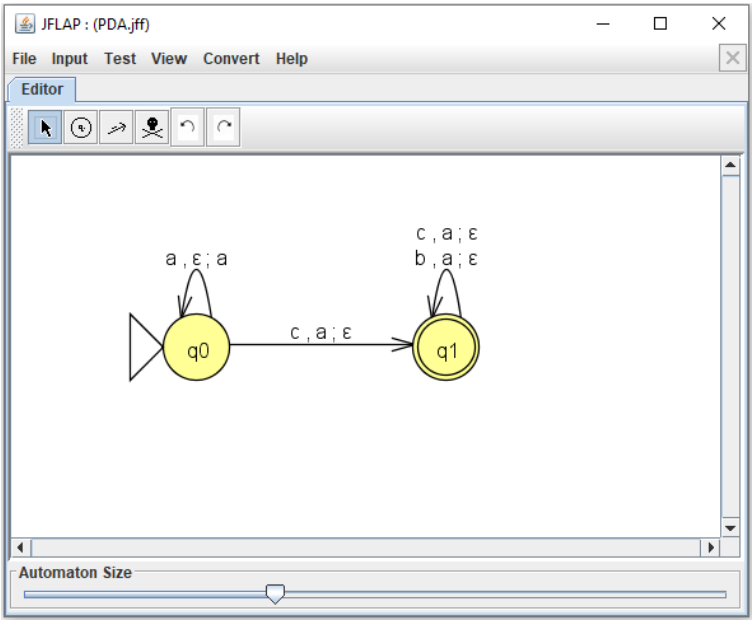

Рис. 1.29. Магазинний автомат

1.5.3. Формальні граматики

Робота з граматиками починається, власне, з введення граматики, як це видно у лівому вікні на рис.1.30.

Меню `Input` надає можливість виконати розбір вхідного рядка (рядків) різноманітними методами, такими як метод грубої сили (Brute Force), контрольований користувачем, Кока-Янгера-Касами (CYK) тощо.

Розбір методом грубої сили з побудовою таблиці та дерева розбору вхідного рядка *ababa* представлені на рис. 1.30 та 1.31 відповідно.

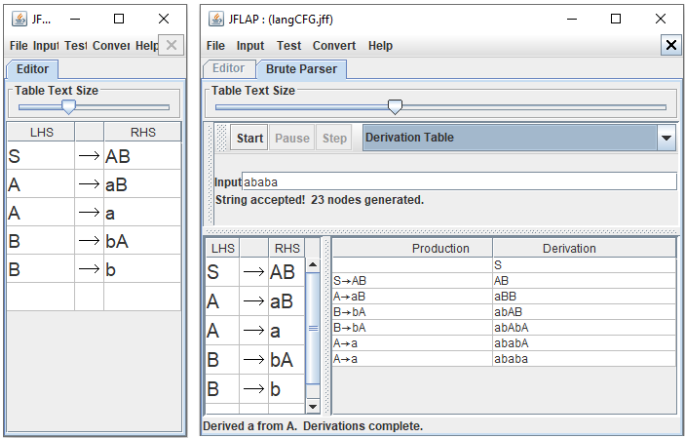

Рис. 1.30. Граматика і таблиця розбору рядка *ababa*

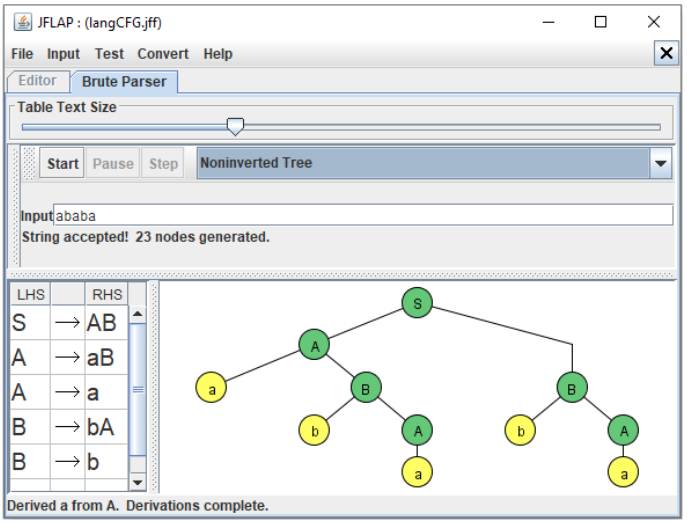

Рис. 1.31. Дерево розбору рядка *ababa*

Розділ 2. Мови програмування

Метамови представлення мов програмування

Мови програмування – можливо, один з найважливіших прикладів застосування формальних мов. Для представлення лексичних, синтаксичних та семантичних аспектів мов програмування, у процесі створення нових та розвитку сотень існуючих, були напрацьовані різноманітні способи їх опису – метамови, які, зазвичай, називають нотаціями.

Повний і точний опис мови програмування, представлений у формі стандарту певного рівня чи іншого офіційного документа, називають специфікацією.

Далі розглядаються популярні (та згадуються специфічні) формальні[2] символьні та графічні способи подання мов програмування.

Зміст і традиційна форма подання специфікацій мов програмування представлена у розділі 2.4

2.1. Лексика

Лексика мов програмування – найпростіша складова, яка належить до класу регулярних (праволінійних, автоматних) мов. Тому лексику мови програмування визначають за допомогою праволінійних генеративних граматик або регулярних виразів, а розпізнають шляхом застосування детермінованих скінченних автоматів.

Зважаючи на те, що регулярні мови є підможиною контекстно вільних мов $L_{\text{регулярні}} \subset L_{\text{КВ}}$, див. розділ 1.3.4, а останні здатні представити більшу частину синтаксису будь-якої мови програмування, то для визначення мови програмування будують єдину контекстно вільну граматику, яка містить також і лексичну складову.

2.2. Синтаксис

Отже, для представлення синтаксису і лексики мов програмування, які належать до класу контекстно вільних мов, використовують технічно зручні метамови, які засновані на так званій нотації Бекуса-Наура. Інша синонімічна назва – форми Бекуса-Наура (БНФ), або, у англомовній редакції, – Backus-Naur

[2] Формальні засоби доповнюються неформальним описом природною мовою, особливо семантична складова

form, Backus normal form (BNF). Кілька розширень БНФ називають розширеною нотацією Бекуса-Наура (РБНФ), або, англійською, Extended BNF (EBNF). Наявність численних версій призводить до необхідності в кожній специфікації, передусім, декларувати у явній формі застосовану метамову.

Серед графічних форм представлення граматик найчастіше використовують синтаксичні діаграми (діаграми Вірта), інколи, і останнім часом дуже рідко, – дерева, які називають при цьому синтаксичними деревами.

Разом з тим дерева дуже часто використовують для представлення дерев виведення (можливо, називаючи їх також конкретними синтаксичними деревами, або просто синтаксичними деревами) та їх редукованих версій (які можуть називати абстрактними синтаксичними деревами, або теж просто синтаксичними деревами). Така термінологічна неузгодженість вимагає уточнення використання значень термінів у кожному конкретному тексті.

Всі ці нотації добре відомі розробникам та користувачам мов програмування, і широко використовуються для точного та наочного представлення синтаксису (і лексики) мов програмування.

2.2.1. Нотація Бекуса-Наура

Нотація Бекуса-Наура передбачає використання таких метасимволів:

- кутових дужок < та > для позначення нетерміналів;
- ::=, що читається "визначається як" чи "дорівнює за означенням";
- вертикальної риски |, яка означає альтернативу.

Приклад. Наведемо граматику, що визначає константу деякої мови програмування. Для зручності обговорення пронумеруємо правила:

```
1  <Константа> ::= <Ціле число> | <Дійсне число>
2  <Ціле число> ::= <Знак> <Беззнакове ціле>  | <Беззнакове ціле>
3  <Дійсне число> ::= <Знак> <Беззнакове дійсне> | <Беззнакове дійсне>
4  <Знак> ::= + | -
5  <Беззнакове ціле>   ::= <Цифра> | <Беззнакова ціле> <Цифра>
6  <Беззнакове дійсне> ::= . <Беззнакове ціле> | <Беззнакове ціле> .
                         | <Беззнакове ціле> . <Беззнакове ціле>
7  <Цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Часто текст граматики форматують, покращуючи сприйняття граматики, наприклад, так:

```
1  <Константа>    ::= <Ціле число>
                     | <Дійсне число>

2  <Ціле число>   ::= <Знак> <Беззнакове ціле>
                     | <Беззнакове ціле>
```

```
3  <Дійсне число> ::= <Знак> <Беззнакове дійсне>
                    | <Беззнакове дійсне>

4  <Знак>         ::= +
                    | -

5  <Беззнакове ціле>   ::= <Цифра>
                         | <Беззнакова ціле> <Цифра>

6  <Беззнакове дійсне> ::= . <Беззнакове ціле>
                         | <Беззнакове ціле> .
                         | <Беззнакове ціле> . <Беззнакове ціле>

7  <Цифра>        ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Отже, тут константа, як це видно з правила 1, є або цілим, або дійсним числом. Ціле число, див. продукцію 2, визначається як `<Беззнакове ціле>` зі знаком, чи без знаку перед ним. Знак, у свою чергу – правило 4, це термінал `+`, або термінал `-`. Продукція 5 визначає (ліворекурсивно) `<Беззнакове ціле>` як одну, або кілька цифр. `<Цифра>` визначається через множину терміналів у продукції 7. `<Дійсне число>` визначається як `<Беззнакове дійсне>` зі знаком чи без знаку перед ним, див. правило 3. Нарешті `<Беззнакове дійсне>` визначається як непорожній ланцюжок, в якому є одна крапка (символ `.`), а решта — цифри. Крапка може бути префіксом ланцюжка, суфіксом, чи розміщуватись у іншій позиції, див. правило 6.

Мові з такою граматикою належать, наприклад, такі ланцюжки (ще кажуть "літерали") `142, - 23, +6789, -.345, +56.9802, 8765`.

Підсумовуючи, можна зазначити, що наведена граматика містить:

- термінали — арабські цифри від 0 до 9, символи `+, -, .`;
- нетермінали — `<Константа>, <Ціле число>, <Дійсне число>, <Знак>, <Беззнакове ціле>, <Беззнакове дійсне>, <Цифра>`;
- ліворекурсивну продукцію 5.2 (і тому наведена граматика константи є ліворекурсивною і ліволінійною). Якби продукція 5 була представлена у формі `<Беззнакове ціле> ::= <Цифра> | <Цифра> <Беззнакова ціле>`, то і продукція 5.2, і граматика були би праволінійними.

Приклад. Граматика оператора (інструкції) присвоювання. Оператор присвоювання, позначимо лексемою `:=`, див. правило 1.

```
1   <Присв>         ::= <Ідентифікатор> := <Вираз>

2   <Ідентифікатор> ::= <Літера>
                      | <Ідентифікатор> <Літера>
                      | <Ідентифікатор> <Цифра>
```

```
3   <Вираз>          ::= <Доданок>
                       | <Вираз> + <Доданок>
                       | <Вираз> - <Доданок>

4   <Доданок>        ::= <Множник>
                       | <Доданок> * <Множник>
                       | <Доданок> / <Множник>

5   <Множник>        ::= <Ідентифікатор>
                       | <Константа>
                       | ( <Вираз> )

6   <Літера>         ::= a | b | c | d | e | f | g | h | i | j
                       | k | l | m | n | o | p | q | r | s | t
                       | u | v | w | x | y | z

7  <Цифра>           ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Нетермінал <Ідентифікатор> традиційно визначимо як ланцюжок, префіксом якого є літера, а решта символів – літери або цифри. Нетермінали <Літера> та <Цифра> визначаються через термінали у продукціях 6 та 7. Арифметичний вираз, представлений нетерміналом <Вираз>, визначає, що окремий доданок, чи сума виразу та доданка, чи добуток виразу та множника теж є виразом, див. правило 3. Нетермінал <Доданок> може бути представлений як окремий множник, чи добуток доданка та множника, або частка від ділення доданка на множник, див. правило 4. При цьому нетермінал <Множник> — це або ідентифікатор, або константа (визначені у попередньому прикладі), або вираз у дужках, див. правило 5.

Приклади, ланцюжків такої мови можуть виглядати так:

```
a:=142 - 23
b1:=(6789+a)/.345*(x-354)
```

Отже, наведена граматика містить:

- термінали — цифри 0 – 9, символи +, -, ., :=, *, /,(,);
- нетермінали — <Присв>, <Ідентифікатор>, <Вираз>, <Доданок>, <Множник>, <Літера> та <Константа>, <Ціле число>, <Знак>, <Дійсне число>, <Беззнакове ціле>, <Беззнакове дійсне>, <Цифра> через використання граматики для нетермінала <Константа>;
- ліворекурсивні продукції 2, 3, 4 та ліворекурсивну граматику для нетермінала <Константа> (і тому наведена граматика інструкції присвоєння, нетермінала <Присв>, є ліворекурсивною).

Приклад. Визначимо граматику простої (Pascal - подібної) мови програмування, яку називатимемо MP_1 і яка надалі знадобиться нам для розгляду необхідних трансформацій.

Кожна програма мовою MP_1 починається з термінала `program`, після якого послідовно розміщуються: ідентифікатор програми, термінал `var`, список оголошень, термінал `begin`, інструкції та термінал `end.` (з крапкою), що завершує програму, див. продукцію 1.

```
1   <Програма> ::= program <Ім'я програми>
                   var <Список оголошень>
                   begin  <Інструкції>  end.

2   <Ім'я програми> ::= <Ідентифікатор>

3   <Ідентифікатор> ::= <Літера>
                      | <Ідентифікатор> <Літера>
                      | <Ідентифікатор> <Цифра>

4   <Список оголошень> ::= <Оголошення>
                      | <Список оголошень> ; <Оголошення>

5   <Оголошення> ::= <Список ідентифікаторів> : <Тип>

6   <Список ідентифікаторів> ::= <Ідентифікатор>
                      | <Список ідентифікаторів> , <Ідентифікатор>

7   <Тип>       ::= integer
                      | real

8   <Інструкції> ::= <Інстр>
                      | <Інструкції> ; <Інстр>

9   <Інстр>     ::= <Присв>
                      | <Чит>
                      | <Вив>
                      | <Цикл>
                      | <Розгалуж>

10  <Присв> ::= <Ідентифікатор> := <Вираз>

11  <Вираз>     ::= <Доданок>
                      | <Вираз> + <Доданок>
                      | <Вираз> - <Доданок>

12  <Доданок>   ::= <Множник>
                      | <Доданок> * <Множник>
                      | <Доданок> / <Множник>

13  <Множник>   ::= <Ідентифікатор>
                      | <Константа>
```

```
                | '(' <Вираз> ')'

14  <Чит> ::= read ( <Список ідентифікаторів> )

15  <Вив> ::= write ( <Список ідентифікаторів> )

16  <Цикл> ::= for <Інд. вираз> do <Блок інструкцій>

17  <Інд. вираз> ::= <Ідентифікатор> :=  <Вираз>
                        to <Вираз>  step <Вираз>

18  <Блок інструкцій> ::= <Інстр>
                | begin <Інструкції> end

19  <Константа> ::= <Ціле число> | <Дійсне число>

20  <Ціле число> ::= <Знак> <Беззнакове ціле>
                | <Беззнакове ціле>

21  <Дійсне число> ::= <Знак> <Беззнакове дійсне>
                | <Беззнакове дійсне>

22  <Знак> ::= + | -

23  <Беззнакове ціле> ::= <Цифра>
                | <Беззнакова ціле> <Цифра>

24  <Беззнакове дійсне> ::= . <Беззнакове ціле>
                | <Беззнакове ціле> .
                | <Беззнакове ціле> . <Беззнакове ціле>

25  <Літера> ::= a | b | c | d | e | f | g | h | i | j
              | k | l | m | n | o | p | q | r | s | t
              | u | v | w | x | y | z

26  <Цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Нетермінал `<Список оголошень>` рекурсивно визначається як одне, або більше оголошень, розділених терміналом `;` (крапка з комою), див. правило 4. Нетермінал `<Оголошення>` через `<Список ідентифікаторів>` визначається як послідовність розділених комою (термінал `,`) ідентифікаторів з наступною двокрапкою (термінал двокрапка `:`) та нетерміналом `<Тип>`, див. правила 5 та 6. Нетермінал `<Тип>`, див. правило 7, визначається або як рядок символів (термінал) `integer`, або як термінал `real`.

Засоби керування потоком обчислень визначаються нетерміналом `<Інструкції>`, правило 8, або як одна інструкція (нетермінал `<Інстр>`), або список (послідовність) інструкцій, розділених символом `;` (крапка з комою). Граматика

передбачає п'ять типів інструкцій: присвоєння, введення, виведення, циклу та розгалуження — нетермінали <Присв>, <Чит>, <Вив>, <Цикл> та <Розгалуж> відповідно, див. правило 9.

Граматика для нетермінала <Присв> розглянута у попередньому прикладі.

Нетермінал <Чит> визначається як рядок — термінал read, термінал (, нетермінал <Список ідентифікаторів>, термінал), див. правило 14.

Нетермінал <Вив> — як рядок: термінал write, термінал (, нетермінал <Список ідентифікаторів>, термінал), див. правило 15.

Інструкція циклу визначається як рядок: термінал for, нетермінал <Інд. вираз>, термінал do, нетермінал <Блок інструкцій>, див. правило 16. Індексний вираз <Інд. вираз> визначає діапазон зміни параметра, представленого ідентифікатором <Ідентифікатор>, та крок такої зміни, див. правило 17. Нетермінал <Блок інструкцій> визначається продукцією 18 як єдина інструкція <Інстр>, або список інструкцій (нетермінал <Інструкції>) між терміналами begin та end.

У продукціях 19 – 26 визначаються розглянуті у попередніх прикладах числові константи та набір літер.

Хоча використання лексики рідної мови має безсумнівні переваги, однак часто для найменування елементів граматики обирають англійську лексику, що вважається стандартом де-факто.

```
1   <Program>       ::= program <ProgName>
                           var <DeclarList>
                        begin  <StatementList>  end.

2   <ProgName>      ::= <Ident>

3   <Ident>         ::= <Letter>
                      | <Ident> <Letter>
                      | <Ident> <Digit>

4   <DeclarList>    ::= <Declaration>
                      | <DeclarList> ; <Declaration>

5   <Declaration>   ::= <IdenttList> : <Type>

6   <IdenttList>    ::= <Ident>
                      | <IdenttList> , <Ident>

7   <Type>          ::= integer | real

8   <StatementList> ::= <Statement>
                      | <StatementList> ; <Statement>

9   <Statement>     ::= <Assign>
                      | <Inp>
```

```
                    | <Out>
                    | <ForStatement>
                    | <IfStatement>

10  <Assign>        ::= <Ident> := <Expression>

11  <Expression>    ::= <Term>
                    | <Expression> + <Term>
                    | <Expression> - <Term>

12  <Term>          ::= <Factor>
                    | <Term> * <Factor>
                    | <Term> / <Factor>

13  <Factor>        ::= <Ident>
                    | <Const>
                    | ( <Expression> )

14  <Inp>           ::= read ( <IdenttList> )

15  <Out>           ::= write ( <IdenttList> )

16  <ForStatement>  ::= for <IndExpr> do <DoBlock>

17  <IndExpr>       ::= <Ident> :=  <Expression>
                    to <Expression> step <Expression>

18  <DoBlock>       ::= <Statement>
                    | begin <StatementList> end

19  <Const>         ::= <IntNumb>
                    | <RealNumb>

20  <IntNumb>       ::= <Sign> <UnsignedInt>
                    | <UnsignedInt>

21  <RealNumb>      ::= <Sign> <UnsignedReal>
                    | <UnsignedReal>

22  <Sign>          ::= + | -

23  <UnsignedInt>   ::= <Digit>
                    | <UnsignedInt> <Digit>

24  <UnsignedReal>  ::= . <UnsignedInt>
                    | <UnsignedInt> .
```

```
                    | <UnsignedInt> . <UnsignedInt>

25  <Letter>        ::= a | b | c | d | e | f | g | h | i | j
                      | k | l | m | n | o | p | q | r | s | t
                      | u | v | w | x | y | z

26  <Digit>         ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Зазначимо, що в англомовній традиції нетерміналам `<Доданок>` та `<Множник>` відповідають нетермінали `<Term>` та `<Factor>`.

2.2.2. Розширена нотація Бекуса – Наура

В розширеній нотації Бекуса – Наура нетермінали представлені ланцюжками з використанням цифр та символів підкреслювання _ чи дефісу -. Термінали — рядки у подвійних чи одинарних лапках[1]. Крім того використовуються метасимволи:

1. метасимвол = означає "визначається як" чи "дорівнює за означенням" (як ::= у БНФ);
2. метасимвол вертикальна риска | означає альтернативу, як і в БНФ;
3. метасимволи [та] — для позначення необов'язковості (опціональності) записаного між ними ланцюжка (ланцюжок зустрічається нуль, або один раз);
4. метасимволи { та } — для позначення того, що записаний між ними ланцюжок може бути повторений нуль або більше разів;
5. метасимволи (та) — для групування символів.

Приклад. Розглянемо фрагмент БНФ – граматики мови MP_1, що визначає ціле число, та трансформуємо її у РБНФ – граматику:

```
<IntNumb> ::= <Sign> <UnsignedInt>
            | <UnsignedInt>
<Sign> ::= + | -
<UnsignedInt> ::= <Digit> | <UnsignedInt> <Digit>
```

Змістовно воно означає, що ціле число – це ланцюжок з однієї або більшої кількості цифр, перед якими може бути (а може і не бути) знак. У РБНФ це можна записати так:

```
IntNumb = [Sign] Digit {Digit}
Sign = '+' | '-'
```

[1]Однак часто приймаються додаткові уточнення щодо найменування символів, як-то нетермінали – з великої літери, ланцюжок маленьких літер (в лапках чи без лапок) – термінал, будь-який ланцюжок в лапках – термінал. Такий підхід використовується і в цьому тексті

Або навіть так:

```
IntNumb = ['+' | '-'] Digit {Digit}
```

Якщо `Sign` та `UnsignedInt` зустрічаються в інших правилах граматики, то варто залишити їх визначення:

```
IntNumb = [Sign] UnsignedInt
Sign = '+' | '-'
UnsignedInt = Digit {Digit}
```

Приклад. Представимо граматику мови MP_1 в нотації РБНФ:

```
Program       = program ProgName
                    var DeclarList
                  begin  StatementList  'end.'
ProgName      = Ident
Ident         = Letter {Letter | Digit }
DeclarList    = Declaration {';' Declaration }
Declaration   = IdenttList ':' Type
IdenttList    = Ident {',' Ident}
Type          = integer | real
StatementList = Statement {';' Statement }
Statement     = Assign
              | Inp
              | Out
              | ForStatement
              | IfStatement
Assign        = Ident ':=' Expression
Expression    = Term
              | Expression '+' Term
              | Expression '-' Term
Term          = Factor
              | Term '*' Factor
              | Term '/' Factor
Factor        = Ident
              | Const
              | '(' Expression ')'
Inp           = read '(' IdenttList ')'
Out           = write '(' IdenttList ')'
ForStatement  = for IndExpr do DoBlock
IndExpr       = Ident ':='  Expression to Expression step Expression
DoBlock       = Statement
              | 'begin' StatementList 'end'
Const         = IntNumb
              | RealNumb
IntNumb       = [Sign] UnsignedInt
```

```
RealNumb      = [Sign] UnsignedReal
Sign          = '+' | '-'
UnsignedInt   = NotZero {Digit}
UnsignedReal  = '.' UnsignedInt
              | UnsignedInt '.'
              | UnsignedInt '.' UnsignedInt
Letter        = a | b | c | d | e | f | g | h | i | j
              | k | l | m | n | o | p | q | r | s | t
              | u | v | w | x | y | z
NotZero       = '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
Digit         = '0' | NotZero
```

Варіанти РБНФ

Існує значна кількість прийнятих розширень нотації РБНФ. Найпоширеніші полягають у застосуванні регулярних виразів для визначення ланцюжків символів та використанні, наприклад, таких метасимволів:

1. квадратні дужки у розумінні "один з вказаних/діапазону" ("будь-який з вказаних/діапазону") , наприклад `Digit = [0123456789]`, `Digit = [0-9]` чи `Letter = [a-z]`;

2. символ `^` (кришка) у розумінні "крім вказаних" ("жоден з вказаних"), наприклад `NotZero = [^0]` або `NotLowLetter = [^a-z]`;

3. символ `?` у розумінні "нуль, або один",
наприклад `IntNumb = Sign? UnsignedInt`;

4. символи `*` та `+` у розумінні "нуль чи більше" та "один чи більше" відповідно, наприклад `DeclarList = Declaration (';' Declaration)*`, чи `UnsignedInt = NotZero Digit+`;

5. символ `,` (кома) як роздільник елементів у тілі продукції,
наприклад `Assign = Ident, ':=', Expression`;

6. символ `:` (двокрапка) у розумінні "визначається як", тобто у ролі `=` чи `::=` в БНФ та РБНФ відповідно;

7. символ `.` (крапка) чи `;` (крапка з комою) для означення кінця продукції.

2.2.3. Інші нотації з власними назвами

Широке використання формальних мов для потреб розробки, представлення і застосування мов програмування, форматів даних та протоколів обміну даними спричинило появу значної кількості нотацій, близьких до нотації Бекуса-Наура.

У результаті уніфікації різноманітних варіантів таких нотацій стала Доповнена нотація Бекуса-Наура — Augmented BNF (ABNF), рекомендована для Інтернет-спільноти, див. [7].

Ще одна нотація, нотація ANTLR, є частиною набору інструментів з тією ж назвою. ANTLR — це потужний генератор парсерів, який широко використовується в наукових і виробничих проєктах для створення різноманітних мов. Нотація ANTLR використовується, зокрема, у специфікації мови C# [17].

Ще один чинник еволюції БНФ — посилення репрезентативної здатності нотації у завданнях компіляції. Прикладами таких нотацій є, зокрема, Labelled BNF (LBNF) та Parsing Expression Grammar (PEG), див. [8, 11]. Приклади застосування див. у [9, 12].

Далі наводиться короткий ілюстративний опис названих метамов[1].

ABNF

Основним принципом ABNF є опис формальної системи мови, яка може використовуватися як протокол двонаправленого зв'язку. Такий підхід зумовлений тим, що технічні специфікації Інтернету часто потребують визначення формального синтаксису.

Відмінності між РБНФ і ABNF включають правила іменування, повторення, альтернативи, незалежність порядку та діапазони значень.

Порівняно з РБНФ зазначимо, зокрема, такі особливості:

1. нетермінали називають іменами правил, або просто правилами, або змінними;

2. імена правил (нетермінали) регістронезалежні;

3. термінали визначаються як ланцюжок у подвійних лапках, або як одна чи кілька цифр з явно вказаною основою (`b, d, x` – двійковою, десятковою, шістнадцятковою відповідно) після символа `%` (відсотків). Так `CR = %d13` та `CR = %x0D` еквівалентні визначення символа повернення каретки. Конкатенація нетерміналів позначається метасимволом `.` (крапка), наприклад `CRLF = %d13.10` чи `a = %d13.10`;

4. альтернатива позначається символом `/` (слеш). Так звана "інкрементальна альтернатива" позначається символом `=/`.
   Наприклад, `A = B / C / D / E / F` може бути представлене як

   ```
   A = B / C
   A =/ D
   A =/ E / F
   ```

5. кількість повторів елемента визначається конструкцією `n1*n2Element` – від `n1` до `n2` включно, зокрема

   - `*Element` означає нуль або більше повторів;
   - `*1Element` означає нуль або один повтор, також допускається конструкція `[Element]`;

[1] Використані приклади граматик запозичені з вказаних у підрозділах джерел

- `1*Element` означає один або більше повторів;
- `2*2Element` означає два повтори;
- `2*4Element` означає два, три або чотири повтори;

6. діапазон значень визанчається з використанням дефіса, наприклад пропозиції `OCTAL = %x30-37` та `OCTAL = "0"/"1"/"2"/ "3"/"4"/"5"/"6"/"7"` – еквівалентні.

Офіційною специфікацією ABNF є інтернет-стандарт [7], представлений у RFC 5234.

ANTLR

Нотація ANTLR[1] є варіантом нотації РБНФ, як це можна бачити з прикладу:

```
grammar Expr;


prog : statement+ ;

statement
     : expr NEWLINE
     | ID '=' expr NEWLINE
     | NEWLINE
     ;

expr : expr ('*'|'/') expr
     | expr ('+'|'-') expr
     | INT
     | ID
     | '(' expr ')'
     ;


ID   : ALPHA (ALPHA | DIGIT)+ ;

INT  : DIGIT1 DIGIT+ ;

fragment
ALPHA : [a-zA-Z] ;

fragment
DIGIT : [0-9] ;

fragment DIGIT1 : [1-9] ;
```

[1] Актуальна версія – ANTLR4

```
NEWLINE : '\r'? '\n' ;

WS   : [ \t]+ -> skip ;
```

Разом з тим, як це буває з нотаціями, що походять від конкретних інструментів, вона містить специфічні засоби, як-то засоби для створення та опрацювання кількох вихідних лексичних потоків — каналів, визначення та виконання (під час розбору) семантичних дій, визначення семантичних предикатів для фільтрації застосовних альтернатив тощо.

Зокрема, у останньому рядку прикладу застосована спеціальна директива `-> skip` для видалення із потоку токенів символів пробілу та горизонтальної табуляції. Той же ефект, але з можливістю подальшого їх опрацювання у оголошеному каналі з ім'ям `HIDDEN`, мало би правило `WS : [ \t]+ -> channel(HIDDEN)`.

Нотація ANTLR дозволяє формувати граматику із фрагментів граматик, представлених у кількох файлах. Кожна граматика може містити спільне або роздільне представлення лексики та синтаксису мови. Нетермінали синтаксичної компоненти починаються з рядкової літери, лексичної — з великої. У наведеному прикладі — великими літерами. Ті з лексичних нетерміналів, що мають тільки допоміжне значення, позначаються за допомогою синтаксичного префікса `fragment`.

Деталі та рекомендації щодо застосування див. [16].

LBNF

LBNF-граматика — це граматика на основі нотації БНФ, де кожне правило позначається міткою. Наприклад проста граматика додавання одиниці може бути представлена у формі:

```
EPlus.   Expr   ::= Expr "+" Number ;
ENum.    Expr   ::= Number ;
NOne.    Number ::= "1" ;
```

Мітка використовується для побудови синтаксичного дерева[1], піддере́ва якого задані нетерміналами правила. Інакше, граматика LBNF складається з набору правил, які мають наступну форму:

```
Мітка "."    Категорія  "::="  (Категорія | Термінал)*  ";"
```

або, більш формально,

```
Ident "."    Ident  "::="  (Ident | String)*  ";"
```

Першим ідентифікатором є мітка правила, за якою йде назва категорії. Праворуч від стрілки продукції (:: =) знаходиться список елементів продукції. Елемент - це або символ категорії (нетермінал), або рядок (термінал).

[1]Тут під синтаксичним деревом треба розуміти дерево виведення

Ідентифікатори, тобто імена правил та символи категорій, можуть бути обрані за бажанням, з обмеженнями, встановленими цільовою мовою, наприклад, прийняти, що ідентифікатор — це непорожня послідовність літер, що починається з великої літери.

Розглянемо, наприклад, граматику арифметичних виразів *ArithmExpr*

```
EInt.   Exp2 ::= Integer ;
ETimes. Exp1 ::= Exp1 "*" Exp2 ;
EPlus.  Exp  ::= Exp  "+" Exp1 ;

_.      Exp2 ::= "(" Exp ")" ;
_.      Exp1 ::= Exp2 ;
_.      Exp  ::= Exp1 ;
```

Мітка правила `_` означає, що його використання при розборі нічого не додає до синтаксичного дерева, а, відтак, не створюється конструктор вузла і сам вузол синтаксичного дерева. За наведеною граматикою синтаксичне дерево для `1 * (2 + 3 )` буде представлене у графічній, рис. 2.1, та аналітичній формі:

```
ETimes (EInt 1) (EPlus (EInt 2) (EInt 3))
```

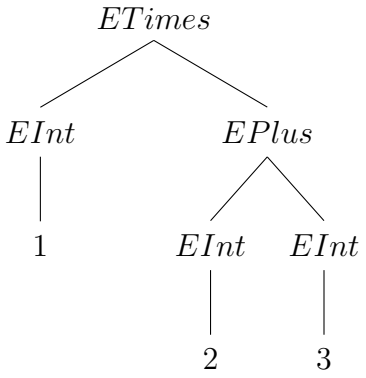

Рис. 2.1. Синтаксичне дерево виразу $1 * (2 + 3)$ в граматиці *ArithmExpr*

Отже, дерево не містить додаткових вузлів, що відповідають останнім трьом продукціям.

Якби четверте правило мало мітку, напр. *EPar*, то дерево мало би графічну, рис. 2.2, та аналітичну форму:

```
ETimes (EInt 1) (EPar (EPlus (EInt 2) (EInt 3)))
```

і, очевидно, не додавало би жодної корисної інформації.

Деталі та рекомендації щодо застосування див. [8, 10].

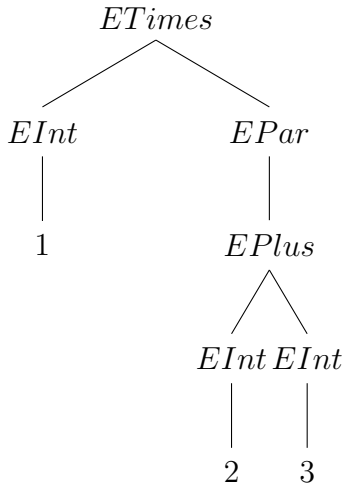

Рис. 2.2. Синтаксичне дерево виразу $1 * (2 + 3)$ в граматиці з міткою EPar

PEG

Нотація PEG (Parsing Expression Grammar, граматика синтаксичного розбору виразів) — зосереджена на представленні контекстно вільних граматик у формі, зручній для низхідного синтаксичного розбору. Зокрема, PEG не дозволяє ліворекурсивних продукцій.

У PEG використовуються одинарні та подвійні лапки для представлення літералів, квадратні дужки для визначення класу символів та круглі дужки для групування символів. Для позначення будь-якого окремого символа використовують крапку (`.`). Постфіксні оператори `*, +, ?` означають нуль або більше, один або більше, нуль або один символ відповідно. Впорядкована альтернатива позначається символом `/`. Префіксні оператори `&` та `!` називають *And*– та *Not*–предикатами відповідно.

У випадку множинності варіантів синтаксичного розбору вхідного рядка PEG передбачає відмінні пріоритети можливих альтернатив. Так, якщо з двох альтернатив перша виявилась успішною, то друга альтернатива взагалі розглядатися не буде.

Наприклад, якщо в РБНФ визначені два правила:

```
A = a b | a
B = a | a b
```

то кожне з них розпізнає мову {a, ab}, тобто вказані продукції еквівалентні.

Натомість дві продукції PEG

```
A ← a b / a
B ← a / a b
```

розпізнають різні мови: перша продукція розпізнає мову {a, ab}, друга — мову {a}. Ефект зумовлений тим, що перша альтернатива (`B ← a`) спрацює завжди, а тому друга (`B ← ab`) ніколи не активізується, і вхідне слово `ab` не буде розпізнане.

Класичну контекстно вільну мову $L_1 = \{a^n b^n | n > 0\}$ засобами PEG можна представити продукцією `S` $\leftarrow$ `a S? b`.

Предикати дозволяють "заглядати вперед" ("lookahead") на довільну глибину вхідного рядка. Це дозволяє представляти мови, які не є контекстно вільними. Наприклад, класична контекстно залежна мова $L_2 = \{a^n b^n c^n | n > 0\}$ може бути визначена граматикою G_2 з правилами

```
S ← &(A !b) a+ B !.
A ← a A? b
B ← b B? c
```

Правила для нетерміналів `A` і `B` здатні розпізнати мову L_1, тобто слова $a^{n_1} b^{n_1}$ і $b^{n_2} c^{n_2}$ відповідно.

Правило для нетермінала `S` буде успішним при розборі вхідного рядка `a+ B !.` (виду $a^k b^{n_2} c^{n_2}$), але тільки за умови, що попередній перегляд ("lookahead") виявив у вхідному рядку слово `&(A !b)`, розпізнане нетерміналом `A` стільки разів, щоб прочитати усі літери `b`, тобто $a^{n_1} b^{n_1}$. Тобто така граматика зможе успішно розпізнати слово, у якому кількість літер визначається співвідношеннями $a^k b^{n_2} c^{n_2}$ та $a^{n_1} b^{n_1}$, з чого, очевидно, $n_1 = k = n_2$, тобто $n_1 = n_2 = n$. Отже, граматика G_2 визначає мову $L_2 = \{a^n b^n c^n | n > 0\}$.

Як це видно з наведених прикладів, репрезентативна здатність PEG достатня для опису не тільки контекстно вільних, але і деяких контекстно залежних мов.

Прикладом останніх застосувань PEG є новий PEG-парсер для мови CPython, див. [12].

2.2.4. Синтаксичні діаграми

Синтаксичні діаграми використовують для графічного подання правил граматики мов програмування[1]. Синтаксична діаграма будується для кожного правила граматики у формі орієнтованого графа, де вузли, що відповідають терміналам граматики позначаються овалами (або кружечками), а нетерміналам — прямокутниками. Зв'язок між вузлами позначається стрілками — ребрами (дугами).

Кожна діаграма має вхід та вихід — вхідну та вихідну точки, які не обов'язково позначаються графічно. Вхід та вихід діаграм здебільшого зорієнтовані зліва – направо, рідше згори – вниз. Вважається, що ланцюжок належить мові, якщо він описує шлях від входу до виходу.

На рис. 2.3 показана діаграма правила граматики арифметичних виразів

```
Expression = Term
           | Expression '+' Term
           | Expression '-' Term}.
```

Дуги не завжди помічають стрілками, щоб не перевантажувати діаграму деталями. Характер плавних відгалужень та під'єднань забезпечує однозначне визначення спрямованості дуг, див. рис. 2.4.

[1] Вперше запропоновані Ніклаусом Віртом для представлення мови Pascal

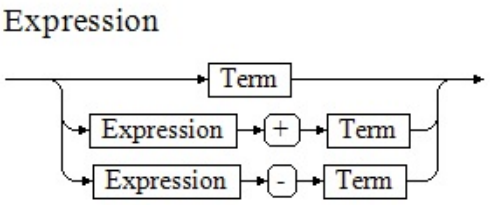

Рис. 2.3. Синтаксична діаграма для Expression

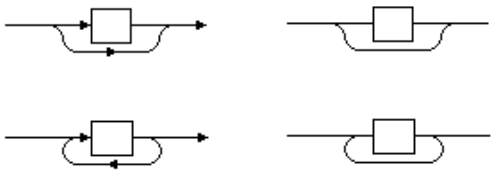

Рис. 2.4. Стрілки та напрямки

Використання синтаксичних діаграм обумовлено необхідністю максимально простого та зрозумілого представлення граматик. Тому для кожного правила будується окрема діаграма. На рис. 2.5 представлена діаграма дійсного числа та тих нетерміналів, через які визначається `RealNumb` за граматикою мови MP_1:

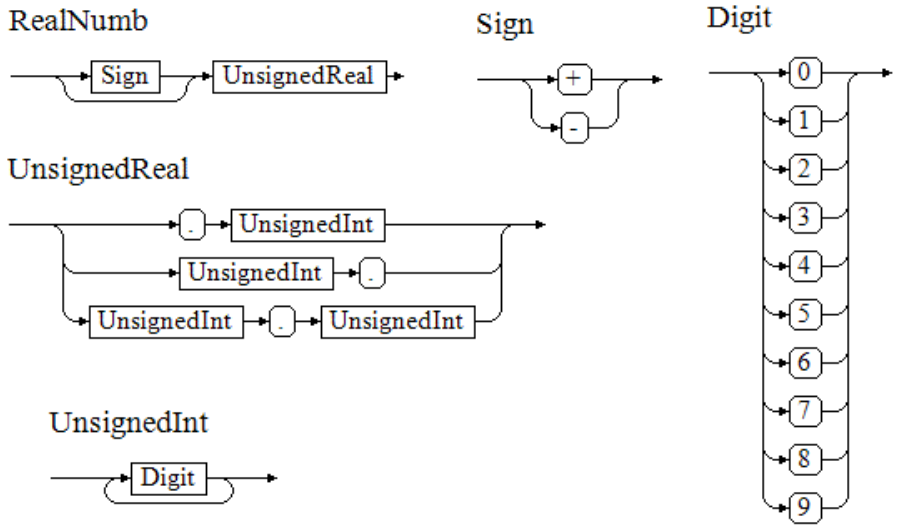

Рис. 2.5. Синтаксичні діаграми кількох нетерміналів мови MP_1

Об'єднана діаграма для `RealNumb`, `Sign`, `UnsignedReal` та `UnsignedInt`, див. рис. 2.6, приводить до все ще придатної для сприйняття діаграми, хоча в ній, можливо, і забагато дрібномасштабних деталей.

Генератори синтаксичних діаграм

Для практичних потреб синтаксичні діаграми будують за допомогою програмних застосунків, на вхід яких подають формальну граматику. Далі наво-

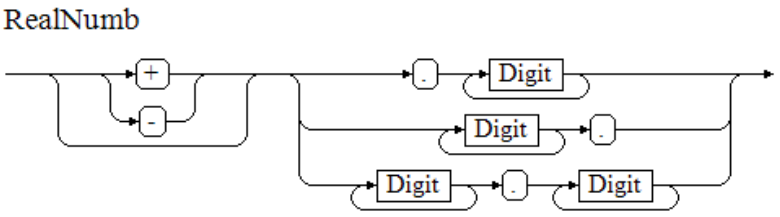

Рис. 2.6. Об'єднана діаграма дійсного числа

диться інформація про декілька таких програм.

EBNF-Vizualiser. Ця програма доступна за посиланням [13] як у вихідних кодах, так і у формі .Net застосунку. На вхід програми подається текстовий файл з розширенням .ebnf, який містить граматику у нотації РБНФ. Програма аналізує правила граматики і візуалізує їх у формі синтаксичних діаграм з можливістю збереження у форматі gif для подальшого використання.

Далі наводиться приклад граматики у нотації РБНФ, яка може бути оброблена програмою `EBNF-Vizualiser` з побудовою синтаксичних діаграм, див. рис. 2.7.

```
prog =   statement {statement} .

statement
     =   expr NEWLINE
     |   ID '=' expr NEWLINE
     |   NEWLINE
     .

expr =   expr ('*'|'/') expr
     |   expr ('+'|'-') expr
     |   INT
     |   ID
     |   '(' expr ')'
     .

ID   =   ALPHA {ALPHA} .
INT  =   DIGIT {DIGIT} .
NEWLINE = '\n' .
WS   =   ' ' | '\t' .
ALPHA = a | b | c | d | e | f | g
      | h | i | j | k | l | m | n
      | o | p | q | r | s | t | u
      | v | w | x | y | z
      .
```

```
DIGIT = '0' | '1' | '2' | '3' | '4'
      | '5' | '6' | '7' | '8' | '9' .
```

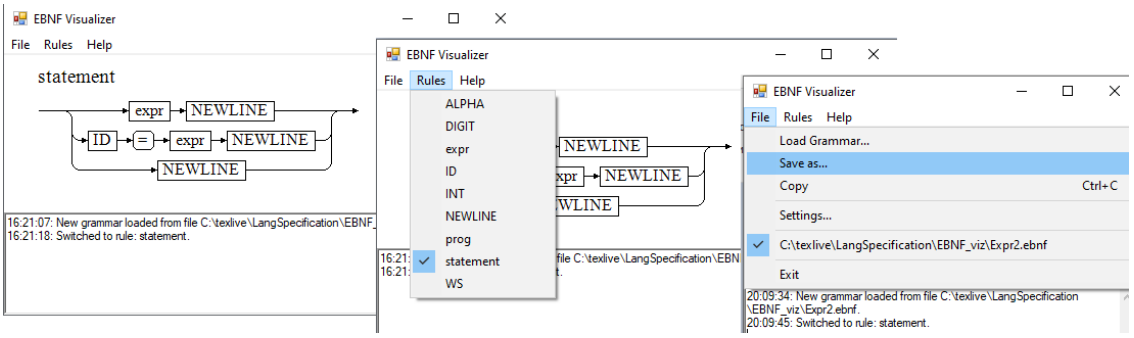

Рис. 2.7. Візуалізатор синтаксичних діаграм

EBNF 2 RailRoad. Цей інструмент, реалізований мовою JavaScript, має інтерфейс командного рядка, призначений для створення документації, включаючи синтаксичні діаграми на основі специфікації ISO/IEC 14977, доступний за посиланням [14]. Дозволяє оптимізувати синтаксичні діаграми, забезпечує швидку навігацію по вихідному документу у html-форматі, підтримує використання markdown-розмітки. Здійснює перевірку граматики на повноту, будує зміст із зазначенням елементів граматики, генерує великі оглядові діаграми тощо.

Далі наводиться приклад граматики у нотації РБНФ, яка може бути оброблена засобами `EBNF 2 RailRoad` з побудовою синтаксичних діаграм у вихідному документі в html-форматі, див. рис. 2.8.

```
prog =   statement, {statement} ;

statement
     =   expr, NEWLINE
     |   ID, '=', expr, NEWLINE
     |   NEWLINE
     ;

expr =   expr, ('*'|'/'), expr
     |   expr, ('+'|'-'), expr
     |   INT
     |   ID
     |   '(', expr, ')'
     ;

ID   =   ALPHA, {ALPHA|DIGIT} ;
INT  =   DIGIT, {DIGIT} ;
NEWLINE = '\n' ;
WS   =   ' ' | '\t' ;
```

```
ALPHA = "a" | "b" | "c" | "d" | "e" | "f" | "g"
      | "h" | "i" | "j" | "k" | "l" | "m" | "n"
      | "o" | "p" | "q" | "r" | "s" | "t" | "u"
      | "v" | "w" | "x" | "y" | "z"
      ;

DIGIT = "0" | "1" | "2" | "3" | "4"
      | "5" | "6" | "7" | "8" | "9"
      ;
```

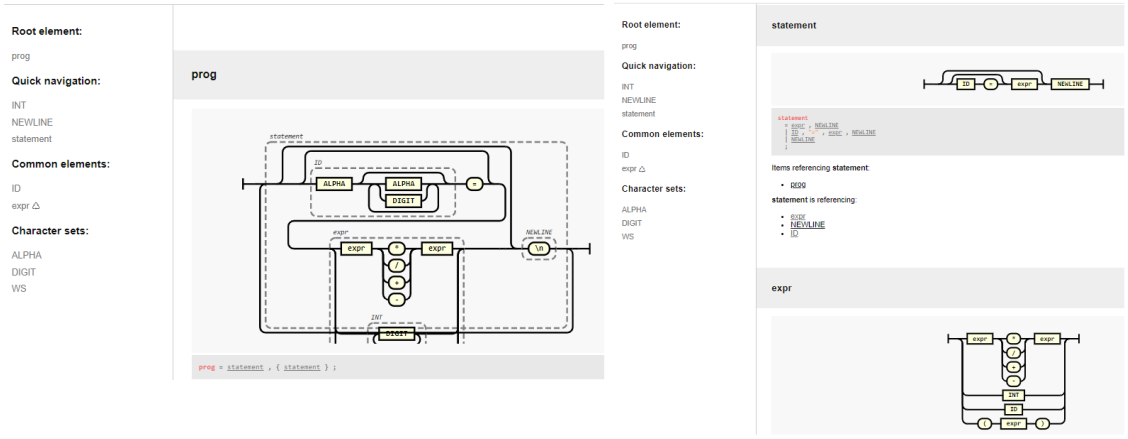

Рис. 2.8. Синтаксичні діаграми у стилі залізничних діаграм

Railroad Diagram Generator. Цей інструмент [15] також забезпечує створення синтаксичних діаграм РБНФ-граматик у стилістиці залізничних діаграм. Особливості цього інструменту полягають у використанні рекомендованої консорціумом W3C (World Wide Web Consortium) нотації РБНФ, збиранні граматик зі специфікацій W3C, їх онлайн редагуванні, представленні діаграм у форматі svg, та реалізації мовами веб-розробки, такими як XQuery, XHTML, CSS, JavaScript.

Далі наводиться приклад граматики у нотації РБНФ, яка може бути оброблена засобами `Railroad Diagram Generator` з побудовою синтаксичних діаграм у вихідному документі в html-форматі, див. рис. 2.9.

```
prog ::=   statement+

statement
     ::=   expr NEWLINE
     |   ID '=' expr NEWLINE
     |   NEWLINE

expr ::=   expr ('*'|'/') expr
     |   expr ('+'|'-') expr
     |   INT
```

```
     |   ID
     |   '(' expr ')'

ID  ::=   ALPHA (ALPHA|DIGIT)*
INT ::=   DIGIT+

NEWLINE ::= '\n'

WS  ::=   ' ' | '\t'

ALPHA ::= "a" | "b" | "c" | "d" | "e" | "f" | "g"
      | "h" | "i" | "j" | "k" | "l" | "m" | "n"
      | "o" | "p" | "q" | "r" | "s" | "t" | "u"
      | "v" | "w" | "x" | "y" | "z"

DIGIT ::= "0" | "1" | "2" | "3" | "4"
      | "5" | "6" | "7" | "8" | "9"
```

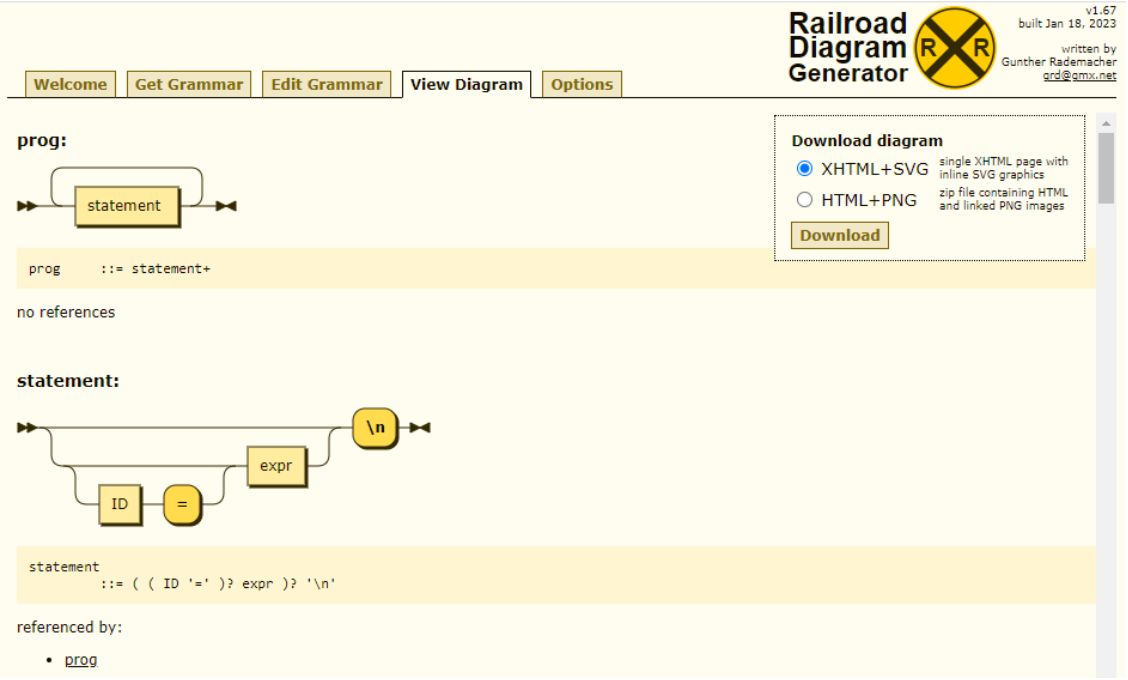

Рис. 2.9. Синтаксичні діаграми, побудовані у `Railroad Diagram Generator`

Вкладка `Welcome` містить самоопис нотації РБНФ, яка використовується за замовчуванням, та посилання для завантаження виконуваної програми і її вихідних кодів для локальної версії інструменту. На вкладці `Get Grammar` наведено список нотацій, які можна використати замість встановленої за замовчуванням. Вкладка `Edit Grammar` дозволяє завантажити файл з граматикою або редагувати її у вікні редагування, або зберегти у локальному файлі. У вкладку `View Diagram` виводяться діагностичні помилки, якщо такі виявлені у граматиці, або синтаксичні діаграми з відповідними правилами граматики. Графічні

представлення нетерміналів та їх ідентифікатори у правилах є посиланнями для переходу до відповідних діаграм. Крім правил, при кожній діаграмі наведено список нетерміналів, що посилаються на поточний.

Visual Studio Code. На рис. 2.10 можна бачити граматику у нотації ANTLR4 та відповідні синтаксичні діаграми, побудовані з використанням розширення `ANTLR4 grammar syntax support`.

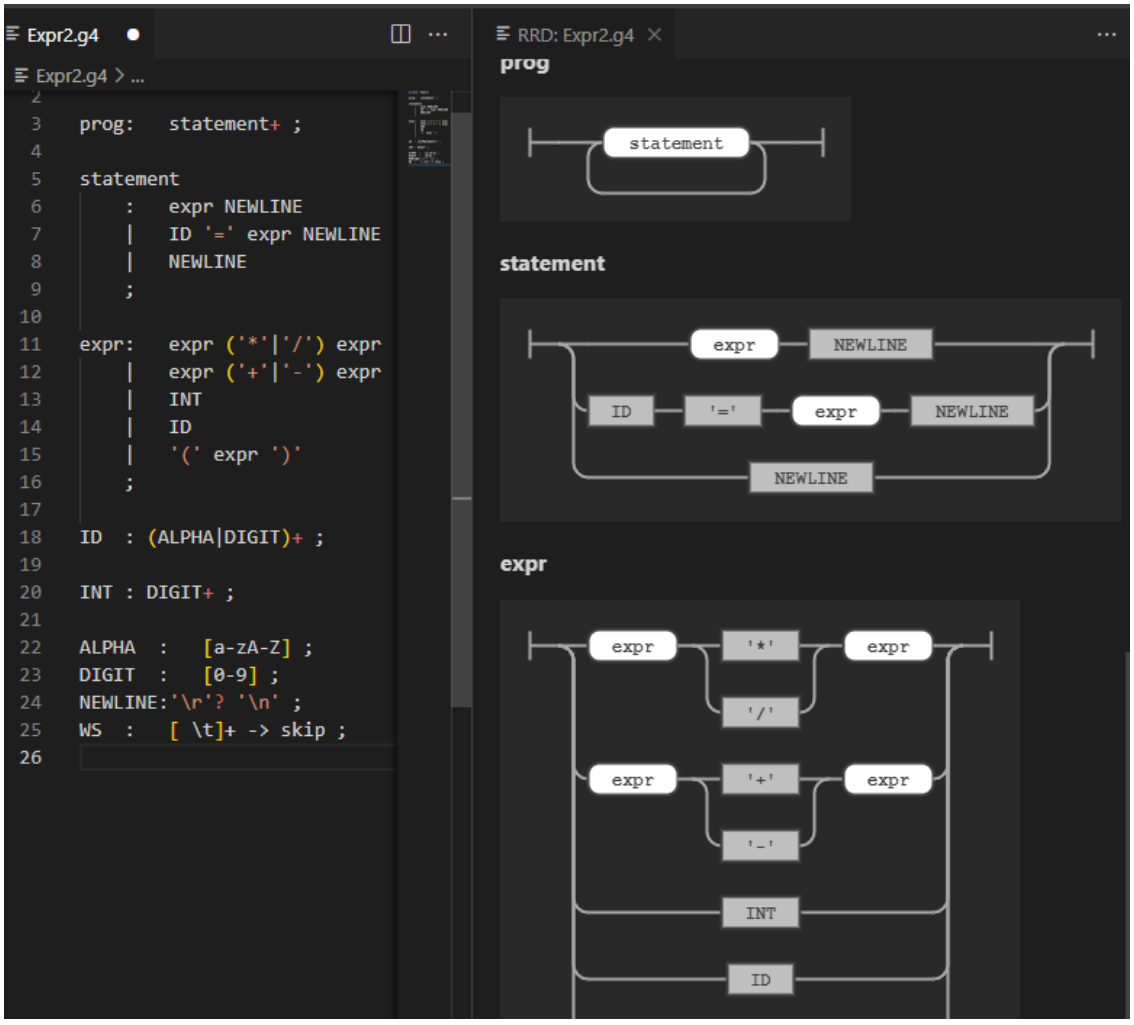

Рис. 2.10. Синтаксичні діаграми для правил ANTLR-граматики

2.2.5. Синтаксичні дерева

Дерева широко використовуються для наочного представлення різноманітних формальних структур — граматик, програм, процесу виведення тощо. Тому термін "синтаксичне дерево" є багатозначним, і в кожному конкретному тексті треба звертати увагу на те, який саме сенс вкладає автор у цей термін.

Для представлення структури граматики

Синтаксична структура граматики може бути представлена за допомогою синтаксичного дерева. Кожен вузол позначається нетерміналом, кількість альтернатив правила визначає кількість нащадків. Листи дерева — термінали, або ланцюжки, структура елементів яких уже представлена у дереві.

```
Integer     = SignedInt
            | UnSignedInt
SignedInt   = Sign UnSignedInt
Sign        = + | -
UnSignedInt = Digit
            | UnSignedInt Digit
Digit       = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

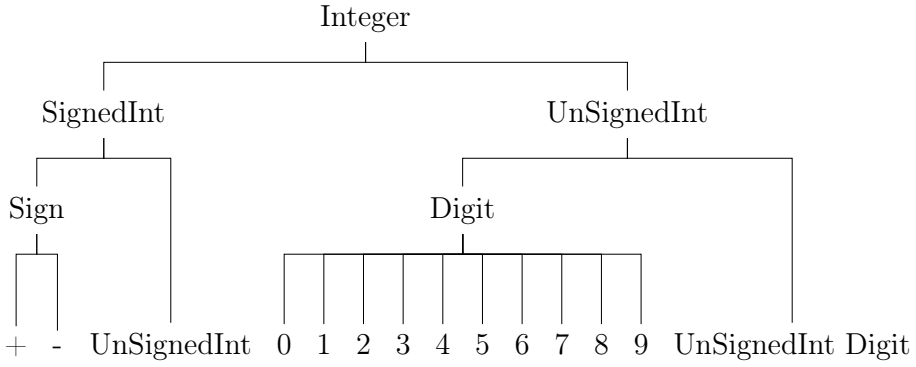

Рис. 2.11. Синтаксичне дерево нетермінала Integer

У граматиці нетермінали `UnSignedInt` і `Digit` зустрічаються у правій частині правил, відповідно тричі та двічі. Але, як видно з рис. 2.11, нащадки кожного з них представляються на діаграмі тільки один раз.

Дерева виведення

Поняття дерева виведення вже було розглянуто на стор. 19. В практиці синтаксичного аналізу дуже часто дерево виведення називають також синтаксичним деревом, можливо з уточнюючими словами. Далі розглядаються конкретне та абстрактне синтаксичні дерева.

Конкретне синтаксичне дерево Якщо кожен використаний при виведенні нетермінал позначається на дереві (внутрішнім) вузлом, то таке дерево називають конктретним синтаксичним деревом. Наприклад, синтаксичний розбір інструкції $a := 12 + 3 * x$ в граматиці

```
Assign = Id := Expr
Expr = Term
     | Term + Term
Term = Factor
```

```
     | Term * Factor
Factor = Id
       |Const
```

приводить до побудови конкретного синтаксичного дерева, див. рис. 2.12.

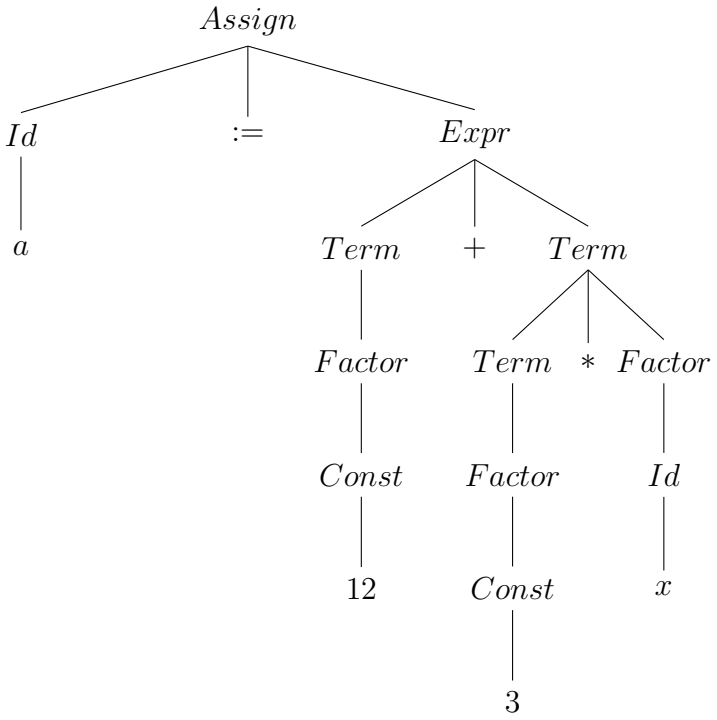

Рис. 2.12. Дерево виведення інструкції $a := 12 + 3 * x$

Часто, якщо частина наявної інформації про виведення сприймається як зайва, замість конкретного синтаксичного дерева будують його редуковану ("спрощену") версію, яку називають абстрактним синтаксичним деревом.

Абстрактне синтаксичне дерево. У абстрактному синтаксичному дереві[1] кожен внутрішній вузол відповідає певній програмній конструкції, див. [1]. Абстрактне синтаксичне дерево, таким чином, протиставляється конкретному синтаксичному дереву (в якому кожен внутрішній вузол — нетермінал).

Так, у абстрактному синаксичному дереві інструкції $a := 12 + 3 * x$, як це видно з рис. 2.13, внутрішні вузли представляють оператори (у тому числі оператор присвоювання :=), а листи — операнди.

2.3. Семантика

Кажучи про "семантику" мови програмування мають на увазі ті її аспекти, які стосуються смислів, або значень, — як кожної конструкції мови окремо, так і у взаємодії з іншими конструкціями. Однозначний і точний опис семантики важливий:

[1] Abstract Syntax Tree, (AST)

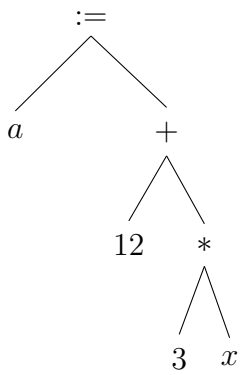

Рис. 2.13. Абстрактне синтаксичне дерево інструкції $a := 12 + 3 * x$

1. розробникам мов програмування – для створення мови програмування з необхідними властивостями;
2. розробникам компіляторів та інтерпретаторів – для розробки реалізації мови, адекватної опису;
3. програмістам – для правильного прогнозування поведінки програми, написаної цією мовою.

Семантика, зазвичай, протиставляється синтаксису та лексиці. Так, конструкція `x := a + x * 2` може бути описана з різних точок зору, наприклад, так:

1. Лексика. Конструкція містить таку послідовність лексем: ідентифікатор `x`, символ присвоювання `:=`, ідентифікатор `a`, оператор додавання `+`, ідентифікатор `x`, оператор множення `*`, цілу константу `2`.
2. Синтаксис. Оператор (statement) присвоювання містить ідентифікатор `x` зліва від знака присвоювання `:=` та арифметичний вираз `a + x * 2` справа від нього.
3. Семантика. Оператор (statement) присвоювання передбачає обчислення значення арифметичного виразу справа від символа присвоювання та збереження результату як значення змінної з ідентифікатором `x`.

Тут, неявно, передбачається, що попередньо у коді були 1) оголошені (якщо це мова з статичною типізацією) та 2) ініціалізовані змінні з ідентифікаторами `x` та `a`, і 3) їх типи визначені, як сумісні з цілим типом. Ці припущення випливають з необхідності застосування арифметичних операторів до своїх аргументів та збереження результату як значення змінної у лівій частині конструкції.

З наведеного видно, що певна смислова складова міститься як на лексичному рівні — про типи лексем (токени чи типи токенів), так і на синтаксичному — про сумісність послідовностей лексем (як про всю конструкцію, так і її частини, наприклад, про арифметичний вираз). Тому опис семантики у специфікаціях мов програмування містить, здебільшого, такі її аспекти, що не представлені однозначно синтаксичними та лексичними засобами.

Наступний приклад ілюструє залежність поведінки однієї і тієї ж програми, в залежності від семантики мови.

2.3.1. Семантика мови і поведінка програми

Нехай у нас є програма, написана певною імперативною мовою з динамічною типізацією.

```
1   a = b+1
2   i = 12
3   j = 3.4
4   s = 'ABC'
5   x1 = s+i+i
6   x2 = i+i+s
7   x3 = s+2*i
8   x4 = 2/0
9   x5 = i + j
10  var c
```

З точки зору лексики, вона не викликає жодних сумнівів, адже все, що вона містить — це ідентифікатори (можливо, `var`, — "ключове слово"), лексеми `=, +, -, *` (використовуються чи не в усіх мовах) та літерали — числові або рядковий ('ABC') .

З боку синтаксису, кожний рядок, за винятком десятого, містить інструкцію присвоювання, тобто ланцюжок — ідентифікатор, лексема `=`, вираз. Синтаксичну коректність десятого рядка можна припустити тільки для деяких мов.

Розглянемо поведінку цієї програми в залежності від семантики мови програмування, тобто від того, як розуміються (тлумачаться), а відтак, і виконуються, одні й ті ж синтаксичні конструкції.

Будемо вважати, що слова "поведінка програми" означає виконання операцій (обчислення), за результатами яких або змінюється зміст пам'яті (присвоювання значень), або генерується повідомлення про помилку.

Для перевірки поведінки наведеної програми у семантиці мов `JavaScript` і `Python` використовувались інтерпретатори Google Chrome[1] і Python[2] відповідно.

З наведеного далі списку, де розглядається кожна конструкція (команда) програми та повідомлення інтерпретатора чи інформація про результат виконання [3], видно, що поведінка програми суттєво залежить від семантики мови.

1. `a = b + 1`
   - Семантика JavaScript: Uncaught ReferenceError: b is not defined at ...
   - Семантика Python: ... NameError: name 'b' is not defined.
2. i = 12
   - Семантика: змінній i присвоюється ціле значення 12.

[1] Версія 109.0.5414.75

[2] Python 3.8.6

[3] Використані інтерпретатори, зустрівши помилку, використовують "режим паніки", див. [1], тобто генерують повідомлення і припиняють виконання програми. Тому, для перевірки реакції на кожну команду прикладу, послідовно закоментовувались рядки з уже виявленими помилками

3. j = 3.4
    - Семантика: змінній j присвоюється дійсне значення 3.4.
4. s = 'ABC'
    - Семантика: змінній s присвоюється значення 'ABC' типу рядок.
5. x1 = s+i+i
    - Семантика JavaScript: змінній $x1$ присвоюється значення '$ABC1212$' типу рядок.
    - Семантика Python: ... TypeError: can only concatenate str (not "int") to str.
6. x2 = i+i+s
    - Семантика JavaScript: змінній $x2$ присвоюється значення '$24ABC$' типу рядок.
    - Семантика Python: ... TypeError: unsupported operand type(s) for +: 'int' and 'str'.
7. x3 = s+2*i
    - Семантика JavaScript: змінній $x3$ присвоюється значення '$ABC24$' типу рядок.
    - Семантика Python: ... TypeError: can only concatenate str (not "int") to str.
8. x4 = 2/0
    - Семантика JavaScript: змінній $x4$ присвоюється значення $Infinity$ числового типу.
    - Семантика Python: ... ZeroDivisionError: division by zero.
9. x5 = i + j
    - Семантика: змінній $x5$ присвоюється дійсне значення 15.4.
10. var c
    - Семантика JavaScript: декларується змінна c, їй присвоюється значення $undefined$ типу $Undefined$.
    - Семантика Python: ... SyntaxError: invalid syntax.

Як це видно з наведеного прикладу, не може бути обчислений вираз, серед операндів якого є невизначена змінна (значення якої невизначене). Більшість виявлених тут відмінностей полягає у наявності (відсутності, особливостях) неявного приведення типів операндів при виконанні операторів, а також

у підтримці чи непідтримці перевантаження операторів (поліморфізму операторів). Нарешті, ділення на нуль трактується як помилка у `Python`, в той час як `JavaScript` знаходить результат — значення $Infinity$ числового типу.

Розглянемо аналогічний приклад програми мовою Pascal[1] — статично типізованою мовою.

```
1  program exmp;
2   var i, j   : integer;
3       i      : integer;
4       x, x2  : real;
5       s, s1  : string[10];
6  begin
7    x := 1;
8    x := 1.5;
9    i := 2.5 + a;
10   i := 2;
11   x1 := 0;
12   x2 := x + i;
13   s := s+1;
14   i := x*2;
15   i :=j;
16   x := s;
17   writeln('x=',x);
18   writeln('i=',i);
19   writeln('j=',j);
20   writeln('x2=',x2);
21 end.
```

Спроба компіляції приводить до повідомлень, які виглядають приблизно так.

```
Compiling main.pas
main.pas(3,12) Error: Duplicate identifier "i"
main.pas(9,14) Error: Identifier not found "a"
main.pas(11,3) Error: Identifier not found "x1"
main.pas(13,8) Warning: Variable "s" does not seem to be initialized
main.pas(13,9) Error: Operator is not overloaded:
                  "ShortString" + "ShortInt"
main.pas(14,9) Error: Incompatible types:
                  got "Real" expected "SmallInt"
main.pas(15,7) Warning: Variable "j" does not seem to be initialized
main.pas(16,8) Error: Incompatible types:
                  got "ShortString" expected "Real"
main.pas(21,4) Fatal: There were 6 errors compiling module, stopping
Fatal: Compilation aborted
...
```

[1]Free Pascal Compiler version 3.0.4, доступний за адресою https://www.onlinegdb.com/

Як бачимо, компілятор відновлюється після помилок (не припиняє роботи після першої), тому аналізується текст усієї програми. Крім повідомлення про уже знайомі з попереднього прикладу помилки, з'явились повідомлення про повторну декларацію змінних та використання неоголошених змінних у лівій частині оператора присвоювання. Додатково — попередження (`Warning`) щодо використання неініціалізованих змінних.

Закоментувавши рядки програми, що містять помилки, здійснюємо компіляцію та виконання програми

```
1  program exmp;
2   var i, j   : integer;
3       // i      : integer;
4        x, x2 : real;
5        s, s1 : string[10];
6  begin
7    x := 1;
8    x := 1.5;
9    //i := 2.5 + a;
10   i := 2;
11   //x1 := 0;
12   x2 := x + i;
13   //s := s+1;
14   //i := x*2;
15   //i :=j;
16   //x := s;
17   writeln('x=',x);
18   writeln('i=',i);
19   writeln('j=',j);
20   writeln('x2=',x2);
21 end.
```

Оскільки помилки усунуті, то серед повідомлень компілятора залишаються тільки попередження щодо використання неініціалізованих змінних та примітки (`Note`) щодо змінних, які були оголошені, але залишились невикористаними.

```
...
Compiling main.pas
main.pas(19,17) Warning: Variable "j" does not seem to be initialized
main.pas(5,6) Note: Local variable "s" not used
main.pas(5,9) Note: Local variable "s1" not used
...
20 lines compiled, 0.1 sec
1 warning(s) issued
2 note(s) issued
x= 1.5000000000000000E+000
i=2
j=0
x2= 3.5000000000000000E+000
```

Зауважимо також, що семантика інструкцій, незважаючи на певну синтаксичну подібність, може суттєво відрізнятись. Так, виконання фрагмента `Python`-програми

```
for i in range(0,2):
    print(i)
```

приводить до результату

```
0
1
```

а аналогічний фрагмент мовою `Pascal`

```
for i:=0 to 2 do
  writeln(i);
```

після виконання, — до результату

```
0
1
2
```

2.3.2. Представлення семантики мови програмування

Семантика може бути представлена або формально, або неформально (у формі природномовного тексту).

Неформальне представлені семантики мови програмування передбачає використання природномовного тексту, яким супроводжують опис синтаксичних конструкцій мови, представлених у БНФ-подібній нотації. Для конкретизації тверджень та роз'яснення опис може доповнюватись прикладами. У розділі 2.3.2 наводиться приклад неформального опису семантики простої імперативної мови `IPL`.

Формальна семантика передбачає побудову певної математичної моделі мови програмування, однак слід зазначити, що єдиного формалізму не існує. Натомість використовуються кілька підходів, і традиційно вважається, що формальна семантика складається із трьох частин (підходів), тобто кажуть про операційну, денотаційну та аксіоматичну семантику. Операційна семантика описує значення мови програмування, вказуючи, як конструкції мови виконуються на абстрактній машині. Денотаційна семантика описує значення мов програмування, використовуючи такі математичні поняття, як повний частковий порядок, неперервна функція, найменша нерухома точка тощо. Аксіоматична семантика описує значення мов програмування, фіксуючи значення програмних сутностей та надаючи для них правила логічного доведення з використанням аксіоматичних теорій, таких як теорія множин та математична логіка.

Для ілюстрації формального підходу у розділі 2.3.2 буде коротко представлено опис елементів простої імперативної мови `IPL` в манері операційної семантики та опис короткої програми цією мовою засобами денотаційної семантики. Матеріали для ілюстративних прикладів були запозичені автором з, уже тепер класичної, книги [4], яку можна вважати гарним стартовим джерелом для детального ознайомлення з формальним описом семантики мов програмування.

Граматика імперативної мови `Imp`.

```
Progr   = Comm

Comm    = scip
        | Id ':=' Aexpr
        | Comm  ';' Comm
        | IfComm
        | WhileComm

IfComm = if Bexpr then Comm else Comm

WhileComm = while Bexpr do Comm

Aexpr   = NAT
        | ID
        | Aexpr '+' Aexpr

Bexpr   = true
        | false
        | Aexpr '==' Aexpr

ID      = LETTER
        | ID (LETTER | DIG)

LETTER  = [A-Za-z]

NAT     = DIG
        | NAT DIG

DIG     = [0-9]
```

Неформальне представлення семантики

Програма, представлена нетерміналом `Progr`, містить одну або більше команд. Виконання програми означає виконання усіх команд програми. Виконання програми завершується нормально, тільки якщо виконання кожної команди завершується нормально, тобто якщо в процесі виконання не було виявлено помилок. Казатимемо, що програма завершилась аварійно, якщо її виконання не завершилось нормально.

Команда `scip` — порожня команда, виконання якої завжди завершується нормально.

По команді `Id ':=' Aexpr` обчислюється значення арифметичного виразу `Aexpr`, а отримане значення зберігається як значення змінної з ідентифікатором `Id`. Виконання завершується нормально, якщо обчислення `Aexpr` завершується нормально.

Арифметичний вираз `Aexpr` — це натуральне число, ідентифікатор, або сума арифметичних виразів.

Якщо при виконанні команди (оператора розгалуження) `IfComm`, поданій тут у формі `IfComm = if Bexpr then Comm1 else Comm2` значення логічного виразу `Bexpr` дорівнює `true`, то виконується команда `Comm1`, інакше (якщо `Bexpr` має значення `false`) — `Comm2`. Виконання `IfComm` завершується нормально тільки якщо обчислення `Bexpr` і виконання `Comm1` (чи `Comm2`) завершуються нормально.

Логічний вираз `Bexpr` — це значення `true`, `false`, або операція порівняння на рівність арифметичних виразів.

Команда (оператор циклу) `WhileComm = while Bexpr do Comm` виконує `Comm` (тіло циклу) нуль або більше разів. Виконання оператора циклу відбувається так:

1. обчислюється логічний вираз `Bexpr`;
2. якщо логічний вираз `Bexpr` має значення `true`, то потік управління передається `Comm`. Коли і якщо виконання тіла циклу `Comm` завершується нормально, потік управління передається на початок оператора циклу, див. пункт 1;
3. якщо логічний вираз `Bexpr` має значення `false`, то потік управління не передається до `Comm`, а виконання оператора циклу `WhileComm` вважається завершенимним нормально.

Формальне представлення семантики

Створюючи формальне представлення семантики мови програмування, ми маємо справу з побудовою математичної моделі. Мета формального опису семантики — надати основу для розуміння та міркування про те, як поводяться програми. Математична модель корисна не тільки для різних видів аналізу та перевірки, але також, на більш фундаментальному рівні, тому, що точне представлення значення програмних конструкцій може виявити неочевидні, але важливі аспекти поведінки програм.

Операційна семантика. Мова IPL передбачає такі синтаксичні множини, — множини ланцюжків символів:

- **N** — натуральні числа і нуль;
- $\mathbf{T} = \{\mathbf{true}, \mathbf{false}\}$ — значення істинності;
- **Identifier** — ідентифікатори змінних;
- **Aexpr** — арифметичні вирази;
- **Bexpr** — логічні вирази;
- **Comm** — команди (оператори).

Для представлення мови приймемо такі символи (можливо з індексами) для позначення ланцюжків символів із оголошених множин:

- $n \in \mathbf{N}$;
- $X, Y \in \mathbf{Identifier}$;
- $a \in \mathbf{Aexpr}$;
- $b \in \mathbf{Bexpr}$;
- $c \in \mathbf{Comm}$.

Тепер можемо записати правила формування елементів мови IPL у БНФ-подібній нотації: Для арифметичних виразів **Aexpr**:

$$a ::= n \mid X \mid a_0 + a_1$$

Для логічних виразів **Bexpr**:

$$b ::= true \mid false \mid a_0 == a_1$$

Для команд (операторів) **Comm**:

$$c ::= scip \mid X := a \mid c_0; c_1 \mid \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1 \mid \textbf{while } b \textbf{ do } c$$

Модель виконання програми, зазвичай, передбачає поняття стану (пам'яті) обчислювача.

Що стосується стану, арифметичний вираз обчислюється як ціле число, а логічний вираз обчислюється як значення істинності. Отримані значення можуть впливати на виконання команд, що призведе до зміни стану. Тому формальний опис поведінки IPL будуватимемо у такій логіці: спочатку ми визначаємо стани, потім обчислення цілих і логічних виразів і, нарешті, виконання команд.

Множина станів Σ містить функцію $\sigma : \mathbf{Identifier} \to N$, а запис $\sigma(X)$ означає значення змінної з ідентифікатором X у стані σ.

Ситуацію, коли арифметичний вираз a у стані σ треба обчислити[1] записують у формі пари (σ, a). При цьому пару (a, σ) називають конфігурацією арифметичного виразу.

Між кожною парою і числом визначають відношення обчислення[2] ($\to$):

$$(n, \sigma) \to n,$$

яке означає, що вираз a у стані σ має значення n.

Відношення оцінювання для чисел, змінних та суми можна визначити через правила:

$$(n, \sigma) \to n,$$
$$(X, \sigma) \to \sigma(X),$$

[1] Кажуть також: виконати оцінку, оцінити (evaluated)

[2] Кажуть також: оцінювання (evaluate)

$$\frac{(a_0, \sigma) \to n_0 \quad (a_1, \sigma) \to n_1}{(a_0 + a_1, \sigma) \to n} \text{ де } n \text{ є сумою } n_0 \text{ і } n_1.$$

В правилах, традиційно, засновок записано над горизонтальною рискою, висновок — під рискою. Аксіоми, — правила з порожнім засновком, записані без горизонтальної риски.

Застосувавши правило для суми при звичайному способі оцінювання значень чисел, можна отримати:

$$\frac{(1, \sigma_0) \to 1 \quad (2, \sigma_0) \to 2}{(1 + 2, \sigma_0) \to 3}.$$

Подібним чином відношення оцінювання для логічних виразів може бути визначене через правила:

$$(true, \sigma) \to true,$$
$$(false, \sigma) \to false,$$

$$\frac{(a_0, \sigma) \to n_0 \quad (a_1, \sigma) \to n_1}{(a_0 == a_1, \sigma) \to true} \text{ якщо } n_0 \text{ дорівнює } n_1,$$

$$\frac{(a_0, \sigma) \to n_0 \quad (a_1, \sigma) \to n_1}{(a_0 == a_1, \sigma) \to false} \text{ якщо } n_0 \text{ не дорівнює } n_1.$$

Зауважимо, що вираз може бути обчислений у певному стані. В той же час програма, а відтак і команди, при виконанні змінюють стан. Виконання програми може завершитися в кінцевому стані або може розійтися і ніколи не досягти заключного стану. Пара (c, σ) представляє конфігурацію команди, яка означає, що треба виконати команду c із стану σ.

Тоді можна визначити відношення виконання $(c, \sigma) \to \sigma'$, яке означає, що (повне) виконання команди c завершується у заключному стані σ'.

Наприклад,

$$(X := 7, \sigma) \to \sigma',$$

або

$$(X := 7, \sigma) \to \sigma[7/X].$$

Такий запис означає, що виконання команди присвоювання приводить до нового стану, де змінна X має значення 7.

Наведемо визначення відношення виконання для команд мови IPL.

Для атомарних команд:

$$(skip, \sigma) \to \sigma$$

$$\frac{(a, \sigma) \to n}{(X := a, \sigma) \to \sigma[n/X]}$$

Для послідовності команд:

$$\frac{(c_0, \sigma) \to \sigma'' \quad (c_1, \sigma'') \to \sigma'}{(c_0; c_1, \sigma) \to \sigma'}$$

Для оператора розгалуження:

$$\frac{(b,\sigma) \to true \quad (c_0,\sigma) \to \sigma'}{(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1,\sigma) \to \sigma'}$$

$$\frac{(b,\sigma) \to false \quad (c_1,\sigma) \to \sigma'}{(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1,\sigma) \to \sigma'}$$

Для оператора циклу:

$$\frac{(b,\sigma) \to false}{(\textbf{while } b \textbf{ do } c,\sigma) \to \sigma}$$

$$\frac{(b,\sigma) \to true \quad (c,\sigma) \to \sigma'' \quad (\textbf{while } b \textbf{ do } c,\sigma'') \to \sigma'}{(\textbf{while } b \textbf{ do } c,\sigma) \to \sigma'}$$

Денотаційна семантика. Використання денотаційної семантики передбачає визначення значення команди як часткову функцію на множині станів Σ.

Арифметичний вираз $a \in Aexpr$ позначає[1] функцію $A[\![a]\!] : \Sigma \to N$. Запис слід розуміти так, що обчислення значення арифметичного виразу a у стані $\sigma \in \Sigma$ приводить до цілого числа $n \in N$.

Логічний $b \in Bexpr$ позначає функцію $B[\![b]\!] : \Sigma \to T$.

Команда c позначає функцію $C[\![c]\!] : \Sigma \to \Sigma$.

Тоді, в термінах денотаційної семантики, програма P з двома інструкціями:

$$X := X + 1;$$

$$Y := Y + X$$

при початкових значеннях змінних $X = 5$ і $Y = 10$, може бути представлена так.

Початковий стан — $\sigma_0[\![X]\!] = 5, \sigma_0[\![Y]\!] = 10$.

Значення програми

$$\begin{array}{rcl} [\![P]\!] & = & C[\![X := X + 1; Y := Y + X]\!]\sigma = \\ & = & C[\![Y := Y + X]\!] \circ C[\![X := X + 1]\sigma = \\ & = & C[\![Y := Y + X]\!]\sigma' = \\ & = & \sigma'' \end{array}$$

де змінні у заключному стані мають значення $\sigma''[\![X]\!] = 6$ і $\sigma''[\![Y]\!] = 16$.

В прикладі було застосовано, зокрема, правило для послідовності операторів.

$$C[\![c_0; c_1]\!] = C[\![c_1]\!] \circ C[\![c_0]\!],$$

де $\circ$ — (правоасоціативний) оператор композиції функцій.

[1] Англійською — denote

2.4. Специфікація мови програмування

Передбачається, що специфікація мови програмування містить точний і повний опис усіх її аспектів. З огляду на практику представлення специфікацій імперативних мов, див. наприклад [17, 18, 19, 20, 21, 22], можна констатувати, що, незважаючи на певні відмінності структури опису, всі вони містять такі елементи:

1. Вступ.
2. Лексична структура:
   - алфавіт;
   - роздільники рядків;
   - пробільні символи;
   - коментарі;
   - ідентифікатори;
   - токени;
   - ключові слова;
   - літерали (цілі, дійсні, логічні, символьні, рядкові, інші).;
   - оператори (operators).
3. Типи даних і змінні:
   - змінні і значення;
   - категорії змінних;
   - базові (примітивні) типи;
   - стандартні типи;
   - сконструйовані типи;
   - приведення типів.
4. Вирази:
   - класифікація;
   - оцінка (обчислення);
   - оператори (operators) — пріоритет, асоціативність, типи операндів і результату.
5. Інструкції (statements):
   - присвоювання значення;
   - циклу;
   - розгалуження;

- інші.

6. Декларації.

7. Структура програми.

8. Повна граматика мови.

Тут доцільно звернути увагу на те, що, скажімо, із понад 800 сторінок (у форматі pdf) специфікації мови `Java` [18], питання, що стосуються контекстно вільних граматик, лексичної структури та синтаксису, займають близько 100 сторінок. Решта 700, тобто понад 87%, складають неформальне представлення семантики (роз'яснення, приклади) та, меншою мірою, прагматики (способів використання).

Крім того, варто зазначити важливість розділу `Вступ`, який містить загальну інформацію як щодо мови програмування, так і специфікації. Тут надається загальна характеристика мови, зазначаються ключові особливості її реалізації, призначення та використання.

Наприклад, у специфікації мови `Java` зазначається [18], що `Java` — це універсальна, конкурентна (concurrent), заснована на класах (class-based), об'єктно-орієнтована мова, суворо та статично типізована.

Із специфікації мови `C#` випливає [17], що `C#` призначена бути простою, сучасною, об'єктно-орієнтованою мовою програмування загального призначення, забезпечувати підтримку принципів розробки програмного забезпечення, таких як сильна перевірка типів, перевірка меж масиву, виявлення спроб використання неініціалізованих змінних і автоматичне збирання сміття.

У Вступі надається також опис нотації, використаної для представлення синтаксису та семантики.

Так, у [19] явно зазначається, що для опису мови `Python` використовується англійська мова, а не формальні специфікації для всього, крім синтаксису та лексичного аналізу, з метою зробити документ більш зрозумілим для звичайного читача, хоч це і залишає місце для неоднозначностей.

У специфікації `Java` нотація представлення граматики описана (в окремому розділі), а специфікація `C#` просто посилається на використання нотації `ANTLR`.

Вступ може містити також історію мови, інформацію про назву та спосіб промовляння назви. Так, назву мови `C#` треба вимовляти як `C Sharp` (`ci шарп`).

Специфікації сучасних мов програмування основного потоку[1] становить сотні сторінок тексту, тож автор вважає недоцільним їх детальний опис, залишаючи це на самостійний розгляд читача, наприклад, за наведеними тут посиланнями.

[1] Англійською — mainstream

Перелік посилань

1. Compilers: Principles, Techniques, and Tools / A. Aho et. al., 2nd edition. - Addison-Wesley Longman Publishing Co., 2006. 1040 p.

2. Hopcroft J. E., Motwani R., Ullman J. D. Introduction to Automata Theory, Languages, and Computation (3rd ed.). Pearson, 2013. 560 p.

3. A. V. Aho and J. D. Ullman, The Theory of Parsing, Translation, and Compiling, Vol. 1, Parsing. Prentice Hall, 1972. 1030 p.

4. Winskel G. The formal semantics of programming languages: an introduction. Cambridge, Massachusetts, London: MIT Press, 1993. 384 p.

5. JFLAP Version 7.1 RELEASED July 27, 2018: веб-сайт. URL: https://www.jflap.org/ (дата звернення: 19.04.2023).

6. Rodger S. H., Finley T. W. JFLAP – An Interactive Formal Languages and Automata Package Jones & Bartlett Publishers, Sudbury, 2005. 212 p. – URL: https://www2.cs.duke.edu/csed/jflap/jflapbook/jflapbook2006.pdf (дата звернення: 19.04.2023).

7. Augmented BNF for Syntax Specifications: ABNF: веб-сайт. URL: https://datatracker.ietf.org/doc/html/rfc5234 (дата звернення: 19.04.2023).

8. LBNF reference: веб-сайт. URL: https://bnfc.readthedocs.io/en/latest/lbnf.html (дата звернення: 19.04.2023).

9. What is the BNF Converter? : веб-сайт. URL: https://bnfc.digitalgrammars.com/ (дата звернення: 19.04.2023).

10. BNF Converter Tutorial : веб-сайт. URL: https://bnfc.digitalgrammars.com/tutorial/bnfc-tutorial.html (дата звернення: 19.04.2023).

11. Ford B. Parsing expression grammars: a recognition-based syntactic foundation. InProceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages 2004 Jan 1 (pp. 111-122).

12. PEP 617 – New PEG parser for CPython : веб-сайт. URL: https://peps.python.org/pep-0617/ (дата звернення: 19.04.2023).

13. EBNF Visualizer: веб-сайт. URL: http://dotnet.jku.at/applications/visualizer/ (дата звернення: 19.04.2023).

14. EBNF 2 RailRoad: веб-сайт. URL: https://github.com/matthijsgroen/ebnf2railroad (дата звернення: 19.04.2023).

15. Railroad Diagram Generator: веб-сайт. URL: https://bottlecaps.de/rr/ui (дата звернення: 19.04.2023).

16. ANTLR: веб-сайт. URL: https://www.antlr.org/ (дата звернення: 19.04.2023).

17. C# 7.0 draft specification: веб-сайт. URL: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/readme (дата звернення: 19.04.2023).

18. The Java® Language Specification, Java SE 20 Edition: веб-сайт. URL: https://docs.oracle.com/javase/specs/jls/se20/html/index.html (дата звернення: 19.04.2023).

19. The Python Language Reference: веб-сайт. URL: https://docs.python.org/dev/reference/index.html (дата звернення: 19.04.2023).

20. Free Pascal Reference guide: веб-сайт. URL: https://www.freepascal.org/docs-html/current/ref/ref.html (дата звернення: 19.04.2023).

21. Pascal Language Reference – URL: https://docs.oracle.com/cd/E19957-01/802-5762/802-5762.pdf (дата звернення: 19.04.2023).

22. Modified Report on the Algorithmic Language Algol 60 – URL: http://www.algol60.org/reports/algol60_mr.pdf (дата звернення: 19.04.2023).