

ТЕОРІЯ ПРОГРАМУВАННЯ

ВСТУП

Філософ Білл Рапапорт зазначив три великі ідеї інформатики^[56]

- Згідно з поглядами [Лейбніца](#), [Буля](#), [Алана Тьюринга](#), [Шеннона](#) і [Морзе](#): є тільки 2 об'єкти, з якими комп'ютер повинен мати справу, щоб представити що завгодно. Іншими словами, тоді уся інформація про довільну обчислювальну проблему може бути представлена з використанням лише 0 і 1 (або іншої бістабільної пари, що може бути тригером між двома легко розрізненими станами, такими як «включено»/«вимкнено», «намагнічено»/«розмагнічено», «висока напруга»/«низька напруга» і т. д.)

Див. також: [Цифрова фізика](#)

- Думка Алана Тьюринга: достатньо 5 дій, щоб комп'ютер міг виконати «що завгодно». Кожен алгоритм може бути висловлений мовою, зрозумілою для комп'ютера у вигляді 5 основних інструкцій [Алана Тьюринга](#):
 - переміщення на позицію ліворуч
 - переміщення на позицію праворуч
 - читання символу на поточній позиції
 - вивести 0 на поточній позиції
 - вивести 1 на поточній позиції

Див. також: [Машина Тьюринга](#)

- Згідно дослідників [Бем^{\[en\]}](#) і Якопіні: є тільки 3 способи комбінування базових дій (в більш складні), які необхідні комп'ютеру, щоб зробити «що завгодно»:

послідовність:

спочатку виконати це; потім зробити оте

вибір:

якщо такий та такий випадок,

THEN (тоді зроби оце)

ELSE (інакше зроби оте)

повторення:

WHILE (до тих пір, поки такий та такий випадок — роби оте)

Якщо перша ідея знайома усім студентам з предмету “Архітектура обчислювальних систем”, а остання з довільного предмету, де вивчається мова програмування, то з другою думкою ми будемо знайомитися лише зараз при вивченні машин Тьюринга.

Інформаційні технології застосовуються зараз практично у всіх сферах життя людства. Ці технології докорінно змінили світ. Значні успіхи в телекомунікаціях, енергетиці, транспорті, медицині, освіті, космічних дослідженнях та інших сферах діяльності людства досягнуто завдяки використанню інформаційних технологій.

Однак таке розширення сфери їх застосувань має і негативну сторону. Людство стає все більш залежним від надійного і правильного функціонування комп'ютерних систем; помилки в їх роботі часом ведуть до величезних фінансових, а навіть і людських втрат. Дотепер фірми-розробники програмного забезпечення не дають гарантію на програмні продукти, які вони пропонують користувачу. Але такого фактично немає в жодній іншій сфері діяльності. Будь-який легальний продавець якогось товару дає гарантію якості і готовий нести (пригадаємо повернення мільйонів автомобілів на заводи виробника у разі виявлення дефектів) фінансову і юридичну відповідальність за поставку недоброякісного продукту. Можна навести і безліч інших прикладів, коли погане програмне забезпечення призводило до знищення космічних ракет, аварій на заводах, гідроелектростанціях тощо. Про складний стан справ говорять і самі фахівці галузі комп'ютерних технологій. У чому ж причини? Вони, безумовно, різноманітні і пов'язані з науковими, економічними, соціальними, психологічними та іншими проблемами створення і використання інформаційних технологій. Серед наукових однією з найважливіших є проблема створення ефективних методів розробки програмних систем. Такі методи мають базуватися на розвиненій теорії програмування, яка має на меті дослідження програм та методів їх аналізу і синтезу. Відзначимо, що тут термін «програма» тлумачиться у широкому сенсі, охоплюючи, зокрема, як програми конкретних мов програмування, так і великі програмні системи.

Даний посібник може розглядатися як вступ до теорії програмування. Ця дисципліна є базовою нормативною дисципліною спеціальності «Інформатика».

Щоб надати початкове уявлення про цю дисципліну, треба визначити її мету і завдання, об'єкт та предмет, а також вказати на її основні методи.

Метою і завданням навчальної дисципліни «Теорія програмування» є засвоєння основних концепцій, принципів та понять сучасного, зокрема композиційного, програмування. У світоглядному аспекті поняття і методи теорії програмування необхідні для обґрунтування та формалізації способів розробки правильних та ефективних програм. У прикладному аспекті апарат теорії програмування необхідний для адекватного моделювання мов специфікацій і програмування та використання побудованих моделей для створення сучасних програмних та інформаційних систем високої якості.

Об'єктом навчальної дисципліни «Теорія програмування» є програми, а її предмет включає в себе вивчення основних (базових) понять теорії програмування, розгляд основних аспектів програм (семантика та синтаксис), їх формалізацію та дослідження, а також уточнення основних методів розробки програм.

Вимоги до знань та вмінь. Для засвоєння матеріалу необхідні знання основ елементарної математики, дискретної математики, алгебри, математичної логіки та теорії алгоритмів. Відповідно в курсі будуть використовуватися методи теорії множин, дискретної математики, універсальної алгебри, математичної логіки і теорії алгоритмів.

Місце у структурно-логічній схемі спеціальності. Нормативна навчальна дисципліна «Теорія програмування» є складовою циклу професійної підготовки фахівців освітньо-кваліфікаційного рівня «бакалавр». Курс теорії програмування потрібен для подальшого вивчення нормативних дисциплін «Системне програмування», «Бази даних та інформаційні системи», «Інтелектуальні системи», «Теорія обчислень», «Штучний інтелект», «Формальні методи розробки програмних систем», «Інформаційні технології», «Прикладна логіка», низки спецкурсів відповідного напрямку.

Суть теорії програмування – категоріальний (поняттєвий) аналіз процесу програмування. Поняття визначаються у єдності їх двох моментів: змісту (інтенціоналу) та обсягу (екстенціоналу). Тому в курсі так багато часу приділяється аналізу основних понять програмування і розкриттю їх інтенціоналів та екстенціоналів. Звідси випливає, що для теорії програмування найважливішим є не

конкретна програма, а відношення між програмними поняттями (тобто не стільки предметна, екстенціональна складова, скільки логічна, інтенціональна). Тому вміння писати конкретні програми ще не означає розуміти теорію програмування. Тут є аналогія з політикою, бо задача політики є не стільки безпосередня робота з пересічною людиною, скільки вибудова суспільних відносин.

Структура курсу:

- формалізація простої мови програмування;
- основні програмні поняття;
- синтактика;
- семантика;
- методи розробки програм.

В першій частині посібника розглядаються основні програмні поняття, формалізується проста мова програмування, вводиться поняття програмної системи, вивчаються формальні мови та граматики, методи уточнення рекурсії програм. У другій частині будуть розглянуті основні підходи до семантики програм та методів їх розробки.

Назва курсу «Теорія програмування» складається з двох термінів – «теорія» та «програмування». Існують різні підходи до побудови теорій. Щоб зрозуміти особливості теорії програмування сформулюємо декілька загальних запитань щодо властивостей теорій.

Яке значення та роль теорії?

Яка теорія потрібна (за побудовою)?

Яким методом будувати теорію?

Коротко відповісти на ці запитання можна наступним чином.

Відповідаючи на перше запитання, відзначимо, що теорію слід розглядати у нерозривному зв'язку з практикою. Дійсно, розробка теорії ґрунтується на науковому дослідженні та аналізі практики. Потім наукові теорії втілюються у практику за допомогою відповідних технологій. Таким чином, маємо цикл: практика – науковедослідження – теорія – технологія – практика.

Відповідаючи на друге запитання, відзначимо, що існують різні підходи до побудови теорій: історичний, логічний, еволюційний тощо. Тут зосередимось на розгляді феноменологічного та сутнісного підходів. Феноменологічний підхід має наметі описати зовнішні прояви певних процесів, а сутнісний – їх внутрішні причини.

Хоча ці підходи мають багато спільного, ми основну увагу будемо приділяти сутнісному підходу.

Третє питання пов'язане з вибором методу побудови теорії. Зважаючи на те, що основні методи побудови програм (систематичний, зверху-вниз, структурний тощо) орієнтовані на поступове уточнення програм, будемо використовувати метод побудови теорії від абстрактного до конкретного (від простого до складного).

Таким чином, основними принципами побудови теорії програмування будуть

- принцип єдності теорії та практики,
- принцип побудови сутнісної теорії,
- принцип побудови теорії методом від абстрактного до конкретного.

Інші принципи будемо обговорювати пізніше.

Автор вдячний студентам факультету кібернетики Київського національного університету імені Тараса Шевченка та студентам Ніжинського державного університету імені Миколи Гоголя за участь у підготовці прикладів.

Закінчення доведення теорем, лем, прикладів або зауважень, позначаємо знаком ■

1. Формалізація простої мови програмування

Центральним завданням теорії, зокрема і теорії програмування, є визначення та дослідження її основних понять. Зробити це не так просто, бо в теоріях, як правило, використовується багато понять, які пов'язані одне з одним. Тому спочатку доцільно ввести основні поняття теорії програмування на невеликому прикладі. Це буде програма знаходження найбільшого спільного дільника двох чисел. Програма записана на деякій простій мові. В цьому розділі ця мова буде формалізована, тобто буде описано її синтаксис і семантику, та їх зв'язок. Також будуть досліджені деякі властивості програм цієї мови.

1.1. Неформальний опис простої мови програмування

Для посилання на певну мову треба надати їй ім'я. Для імені часто використовують аббревіатуру короткої характеристики мови. Зважаючи на традиції використання латинських літер у науковій символіці, утворимо аббревіатуру SIPL від характеристики Simple Programming Language (Проста Мова Програмування). Цю аббревіатуру можна розшифрувати по-іншому, вказуючи на імперативність або структурованість цієї мови:

SIPL – Simple Imperative Programming Language,
SIPL – Structured Imperative Programming Language.

Говорячи неформально, мова SIPL має числа та змінні цілого типу, над якими будуються арифметичні вирази та умови. Основними операторами є присвоєння, послідовне виконання, розгалуження, цикл.

Мова SIPL може розглядатися як надзвичайно спрощена традиційна мова програмування, наприклад, Паскаль. У мові SIPL відсутні складні типи даних, оператори вводу-виводу, процедури та багато інших конструкцій традиційних мов програмування. Також немає явної типізації. Разом із тим ця мова є досить потужною, щоб програмувати різні арифметичні функції, більше того, в цій мові можуть бути запрограмовані всі обчислювані функції над цілими числами. Мета розгляду саме такої мови програмування полягає в тому, щоб формалізація та дослідження цієї мови були якомога простішими.

Приклад 1.1. Програма GCD знаходження найбільшого спільного дільника чисел M та N за алгоритмом Евкліда:

```
GCD ≡  
begin  
  while  $\neg M=N$  do  
    if  $M>N$  then  $M:=M-N$  else  $N:=N-M$   
  end
```

Тут $\neg M=N$ означає $M \neq N$. Оскільки програми призначені для обчислення результатів за вхідними даними, розглянемо неформально процес виконання цієї програми на вхідних даних, у яких M має значення 8, а N – 16. Вважаємо, що дані записуються в пам'ять, а оператори виконуються деяким процесором. Тому розмітимо програму, відмічаючи оператори мітками:

```
0: begin  
  1: while  $\neg M=N$  do  
    2: if  $M>N$  then 3:  $M:=M-N$  else 4:  $N:=N-M$   
  end
```

Процес виконання програми можна подати у вигляді таблиці, кожний рядок якої вказує на номер виконуваного оператора та на нові значення змінних (деталі процесу виконання

можна знайти у книжках з основ програмування). Для нашого прикладу така таблиця може мати наступний вигляд.

Таблиця 1.1

Мітка	Значення умови	Значення M	Значення N
0		8	16
1	$\neg M=N$ – true		
2	$M>N$ – false		
4		8	
1	$\neg M=N$ – false		

Програма припиняє роботу із значеннями M та N, рівними 8. Це число і є найбільшим спільним дільником.

Аналізуючи цю програму та процес її виконання, відзначаємо два аспекти:

- синтаксичний (текст програми);
- семантичний (смісл програми – те, що вона робить).

У нашому прикладі ні синтаксис, ні семантика не задані точно (формально). Ми маємо лише інтуїтивне розуміння програм мови SIPL. Тому важко відповідати на питання, які вимагають точного опису мови. Наприклад незрозуміло, чи можна поставити крапку з комою між оператором та символом end, що дозволяється робити в багатьох мовах програмування (синтаксичний аспект). Незрозуміло також, чи завжди наведена програма буде обчислювати найбільший спільний дільник для довільних значень M та N (семантичний аспект).

Щоб відповідь на такі питання стала можливою, необхідно надати строгі (формальні) визначення нашої мови програмування. Лише тоді можна буде застосувати математичні методи дослідження програм та будувати відповідні системи програмування, зокрема транслятори та інтерпретатори.

Безумовно, уточнення (експлікацію) синтаксичного та семантичного аспектів можна давати по-різному. Спочатку розглянемо традиційні формалізми для опису синтаксису мов.

1.2. Формальний опис синтаксису мови SIPL

Для опису синтаксису мов зазвичай використовують БНФ (Форми Бекуса-Наура). Програми (або їх частини) виводяться із метазмінних (нетерміналів), які записуються у кутових дужках. Метазмінні задають синтаксичні класи. У процесі виводу метазмінні замінюються на праві частини правил, що задають ці метазмінні. Праві частини для однієї метазмінної розділяються знаком альтернативи «|». Процес породження припиняється, якщо всі метазмінні замінено на термінальні символи (тобто символи без кутових дужок).

Синтаксис мови SIPL можна задати за допомогою наступної БНФ.

Таблиця 1.2

Ліва частина	Ім'я	Права частина правила (дефінієнс)	пра-вила
правила – метазмінна			

(дефінієндум)

<програма > ::= begin <оператор > end

NP1

NS1 <оператор > ::= <змінна >:=<вираз > |

NS2 <оператор > ; <оператор > |

NS3 if <умова > then <оператор > else <оператор >

NS4 while <умова > do <оператор > |

NS5 begin <оператор > end |

NS6 skip

NA1- <вираз > ::= <число > | <змінна > | <вираз > + <вираз > |

NA6 <вираз > - <вираз > | <вираз > * <вираз > |

 (<вираз >)

NB1- <умова > ::= <вираз > = <вираз > | <вираз > > <вираз > |

NB5 <умова > V <умова > | ¬ <умова > | (<умова >)

NV... <змінна > ::= . . . M | N | . . .

NN... <число > ::= . . . -1 | 0 | 1 | 2 | 3 | . . .

Наведена БНФ задає мову SIPL як набір речень (слів), які виводяться з метазмінної <програма >. Точне визначення виводу буде подано пізніше, зараз тільки відмітимо, що вивід можна подати у вигляді дерева.

Щоб переконатись у синтаксичній правильності програми, треба побудувати її вивід. Зокрема, для нашої програми GCD маємо дерево виводу, подане на рис. 1.1.

<програма>

begin <оператор>

end

while <умова> do

<оператор>

 ¬

 <умова>

<вираз> = <вираз>

<змінна>

<змінна> if <умова> then

<оператор> else <оператор>

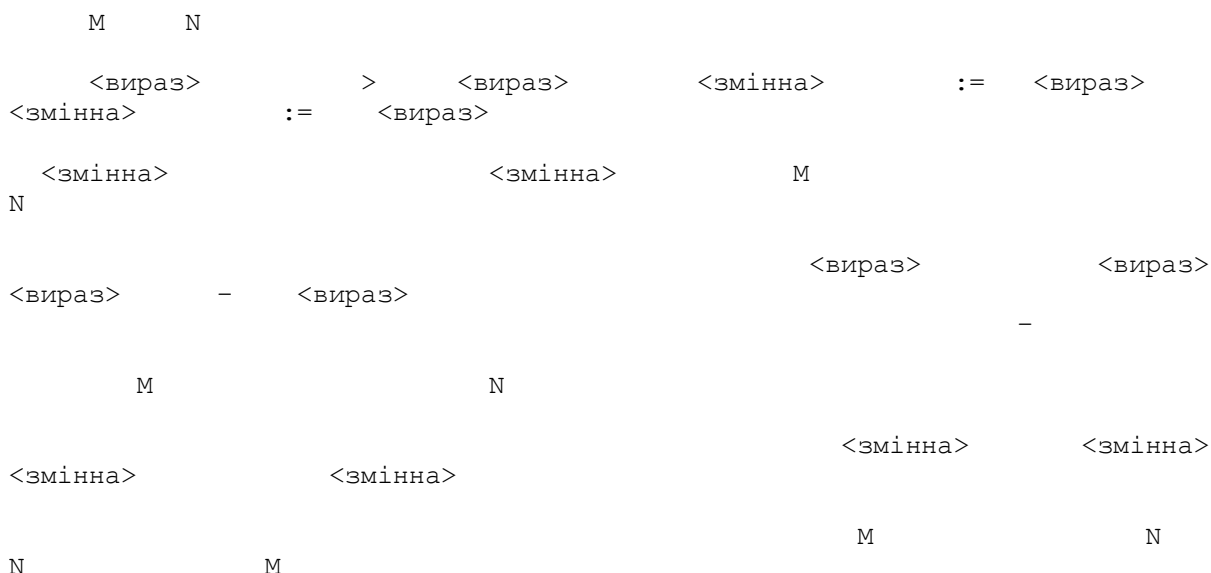


Рисунок 1.1. Дерево синтаксичного виводу програми GCD

Побудоване дерево синтаксичного виводу дозволяє стверджувати синтаксичну правильність програми GCD. Дійсно, якщо записати зліва направо послідовність усіх термінальних символів, то отримаємо програму GCD, що говорить про її вивідність у БНФ мови SIPL, тобто про її синтаксичну правильність.

Аналізуючи далі наведену БНФ можна відзначити, що вона досить просто описує основні конструкції мови SIPL. Але зворотню стороною цієї простоти є неоднозначність цієї БНФ. Наприклад, вираз $X+Y*Z$ має два різних дерева виводу, що ведуть до різних семантик (рис. 1.2). Тому неоднозначність БНФ значно ускладнює семантичний аналіз програм.

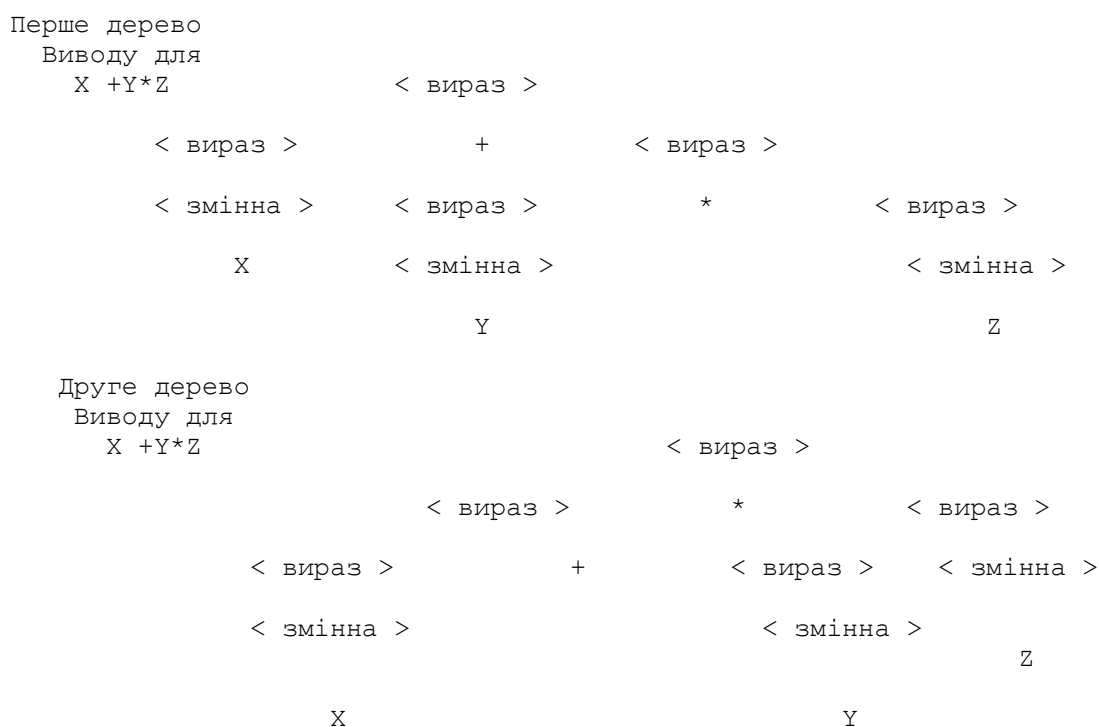


Рисунок 1.2. Деревя синтаксичного виводу виразу $X+Y*Z$

Однозначності побудови дерева синтаксичного виводу можна досягти введенням пріоритетів операцій та правил асоціативності для операцій одного пріоритету. Слід відзначити, що в математиці при описі пріоритетів, як правило, не розрізняють синтаксичний і семантичний аспекти, тому часто говорять, що пріоритети задають порядок виконання операцій (семантичний аспект) замість того, щоб говорити про порядок структурного аналізу виразу (синтаксичний аспект). Такий розгляд об'єкта, коли не виокремлюються його різні аспекти та складові, часто називають синкретичним. У випадку мови SIPL однозначність аналізу може порушуватися при формуванні операндів операцій (синтаксичних конструкторів) різного типу. Будемо виокремлювати чотири типи операцій:

- арифметичні (бінарні операції +, -, *);
- порівняння (бінарні операції =, >);
- булеві (бінарна операція диз'юнкції V та унарна операція заперечення ¬);
- оператори (присвоєння :=, оператор послідовного виконання ;, умовний оператор if_then_else та оператор циклу while_do).

Тут символ підкреслення позначає операнд (або параметр) оператора. Перші три групи операцій є традиційними, як і їх пріоритети. Четверта група потребує певного пояснення. Річ у тім, що, наприклад, синтаксична конструкція виду if b then S1 else S2; S3 може аналізуватися по-різному. Одне трактування може відносити до мовного оператора (його else-частини) послідовність операторів S2; S3, інше трактування - тільки один оператор S2. Різні трактування будуть задавати різну семантику. Те саме стосується і конструкцій виду while b do S1; S2 та S1; S2; S3. На семантику також впливає порядок застосування операцій одного типу. Так, вираз a1-a2-a3 можна аналізувати як зліва направо (лівоасоціативно), що можна подати як (a1-a2)-a3, так і справа наліво (правоасоціативно), що можна подати як a1-(a2-a3) (= (a1-a2)+a3). Щоб подолати неоднозначність синтаксичного аналізу, будемо вважати, що всі бінарні операції лівоасоціативні (обчислюються зліва направо) та що пріоритет операцій наступний (у порядку зменшення):

- 1) множення *;
- 2) додавання + та віднімання -;
- 3) операції порівняння =, >;
- 4) заперечення ¬;
- 5) диз'юнкція V;
- 6) присвоєння :=;
- 7) оператор циклу while_do;
- 8) умовний оператор if_then_else;
- 9) послідовне виконання ;.

Зауважимо ще раз, що однозначність синтаксичного аналізу не зовсім адекватно подається пріоритетами операцій, які мають семантичну природу. Тому можливі розбіжності між цими поняттями.

Є і друга можливість досягти однозначності БНФ мови SIPL - ввести нові метасимволи та нові правила їх визначення. Пропонуємо читачам зробити це самостійно.

Наведені вище позначення метазмінних не зовсім зручні, тому часто використовують форми, більш близькі до математики. Введемо позначення для всіх синтаксичних категорій та нові метазмінні, що вказують на представників цих категорій.

Таблиця 1.3

Метазмінна	Синтаксична категорія
Нова метазмінна	
<програма >	Prog
P	

	<оператор >	Stm
S		
	<вираз >	Aexp
a		
	<умова >	Bexp
b		
	<змінна >	Var
x		
	<число >	Num
n		

У наведених позначеннях БНФ мови SIPL набуває наступного вигляду.

Таблиця 1.4

Ліва частина правила - Ім'я	Права частина правила
метазмінна правила	
P1	P ::= begin S end
S1-S6	S ::= x:=a S ₁ ; S ₂ if b then S ₁ else S ₂ while b do S begin S end skip
A1- A6	a ::= n x a ₁ + a ₂ a ₁ - a ₂ a ₁ a ₂ (a ₁) 1 2 1 2 1 * 2
B1 - B5	b ::= a = a ₁ a ₁ > a ₂ b ₁ V b ₂ ¬b ₁ (b ₁) 1 2 1 2 1 2
NV...	x ::= . . . M N . . .
NN...	n ::= . . . -1 0 1 2 3 . . .

Надалі будемо користуватися введеними позначеннями для запису структури програм та їх складових.

Зауваження 1.3. Таблиці 1.3 та 1.4 надають можливість розглядати наведений формалізм подання синтаксису мову SIPL як певну багатоосновну алгебру (синтаксичну алгебру). Основами цієї алгебри є множини Prog, Stm, Aexp, Bexp, Var та Num, а операціями - перетворення, що задаються таблицею 1.4.

Дотепер ми користувались інтуїтивним розумінням БНФ . Разом із тим слід розуміти, що можуть існувати різні уточнення БНФ (про це буде йтиметься в розділі із синтактики). Зараз приймемо, що БНФ дозволяє прописати індуктивне визначення синтаксичних категорій. Дійсно, наведена БНФ визначає шість синтаксичних категорій (класів): Num, Var, Aexp, Bexp, Stm, Prog. Вони можуть бути задані наступним індуктивним визначенням.

1. База індукції:

- Num={..., - 1, 0, 1, 2, 3, ...};
- Var={M, N, ...};
- якщо $n \in \text{Num}$, то $n \in \text{Aexpr}$;
- якщо $x \in \text{Var}$, то $x \in \text{Aexpr}$;
- skip належить Stm.

2. Крок індукції:

- якщо $a_1, a_2 \in \text{Aexpr}$, то записи (вирази)

- $a_1 + a_2$,

- $a_1 - a_2$,

- $a_1 * a_2$,

- (a)
належать Aexpr,

- якщо $a_1, a_2 \in \text{Aexpr}$, $b_1, b_2 \in \text{Bexpr}$, то записи (умови)

- $a_1 = a_2$,

- $a_1 > a_2$,

- $b_1 \vee b_2$,

- $\neg b_1$,
- (b)
належать Bexpr,

- якщо $x \in \text{Var}$, $a \in \text{Aexpr}$, $b \in \text{Bexpr}$, $S_1, S_2 \in \text{Stm}$, то записи (оператори)

- $x := a$,

- $S_1 ; S_2$,

- if b then S_1 else S_2 ,

- while b do S_1 ,

- begin S_1 end

належать Stm,

- якщо $S \in \text{Stm}$, то запис begin S_1 end належить Prog.

На підставі такого індуктивного визначення може бути задане визначення

синтаксичних категорій за допомогою рекурентних співвідношень. Пропонуємо читачам зробити це самостійно.

Таким чином, можна стверджувати, що наведеними визначеннями синтаксис мови SIPL задано точно (формально).

Перейдемо тепер до визначення семантики мови SIPL.

1.3. Формальний опис семантики мови SIPL

Семантика задає значення (смысл) програми. Наш приклад показує, що смысл програми - перетворення вхідних даних у вихідні. У математиці такі

перетворення називають функціями. Тому до семантичних понять відносять поняття даних, функцій та методів їх конструювання. Такі методи називаються композиціями, а відповідна семантика часто називається композиційною. Будемо вживати термін композиційна семантика, тому що саме композиції і визначають її властивості.

Композиційна семантика є певною конкретизацією функціональної семантики тому, що базується на тлумаченні програм як функцій.

Перейдемо до розгляду композиційної семантики мови SIPL.

1.3.1. Дані

Базові типи даних – множини цілих чисел, булевих значень та змінних (імен):

- $\text{Int} = \{ \dots, -1, 0, 1, 2, \dots \};$
- $\text{Bool} = \{\text{true}, \text{false}\};$
- $\text{Var} = \{\dots, M, N, \dots\}.$

Похідні типи – множина станів змінних (наборів іменованих значень, наборів змінних з їх значеннями):

- $\text{State} = \text{Var} \rightarrow \text{Int}.$

Приклади станів змінних: $[\text{Ma}8, \text{Na}16], [\text{Ma}3, \text{Xa} 4, \text{Ya}2, \text{Na}16].$

Визначимо тепер операції на базових типах даних. Щоб відрізнити операції від символів, якими вони позначаються, їх пишуть жирним шрифтом або вводять спеціальні позначення. Ми тут оберемо другий варіант, вводючи нові позначення для операцій на типах даних.

Операції на множині Int . Символам $+$, $-$, $*$ відповідають операції add , sub , mult (додавання, віднімання, множення). Це бінарні операції типу $\text{Int}2 \rightarrow \text{Int}$.

Операції на множині Bool . Символам \vee , \neg відповідають операції or , neg . Це бінарна операція типу $\text{Bool} \rightarrow \text{Bool}$ (диз'юнкція) та унарна операція типу $\text{Bool} \rightarrow \text{Bool}$ (заперечення).

У мові також є операції змішаного типу. Символам операцій порівняння $=$, $>$ відповідають операції eq , gr . Це бінарні операції типу $\text{Int}2 \rightarrow \text{Bool}$.

Вивчення властивостей даних та операцій у математиці відбувається на основі поняття алгебри. В першому наближенні алгебру можна тлумачити як множину із заданими на ній операціями. Для нашої мови маємо наступні алгебри.

Алгебра цілих чисел : $A_{\text{Int}} = \langle \text{Int}; \text{add}, \text{sub}, \text{mult} \rangle.$

Алгебра булевих значень : $A_{\text{Bool}} = \langle \text{Bool}; \text{or}, \text{neg} \rangle.$

Якщо додати операції порівняння, отримаємо двоосновну алгебру базових даних:

$A_{\text{Int_Bool}} = \langle \text{Int}, \text{Bool}; \text{add}, \text{sub}, \text{mult}, \text{or}, \text{neg}, \text{eq},$

$\text{gr} \rangle.$

Для опису мови SIPL треба додати ще одну основу – множину станів змінних. На цій основі задана бінарна операція накладання ∇ (для цієї операції іноді вживається термін «накладка»). Ця операція за двома станами будує новий стан змінних, до якого входять усі іменовані значення з другого стану та і ті значення з першого стану, імена яких не входять до першого стану. Ця операція подібна до операції копіювання каталогів із спеціальним випадком однакових імен файлів у двох каталогах. У цьому випадку файл із першого каталогу замінюється файлом з тим же іменем з другого каталога.

Наприклад:

$[\text{Ma}8, \text{Na}16] \nabla [\text{Ma}3, \text{Xa} 4, \text{Ya}2] = [\text{Ma}3, \text{Na}16, \text{Xa} 4, \text{Ya}2].$

Уведемо відразу і відношення розширення \subseteq станів новими іменованими значеннями. Наприклад,

$[\text{Ma}8, \text{Na}16] \subseteq [\text{Ma}8, \text{Xa} 4, \text{Ya}2, \text{Na}16].$

Це відношення не включаємо в алгебри, пов'язані з мовою SIPL, бо воно в цій мові безпосередньо не використовується, але буде корисним для доведення властивостей програм цієї мови.

Для створення та оперування із станами змінних треба визначити дві функції: іменування $\Rightarrow x: \text{Int} \rightarrow \text{State}$ та розіменування $x \Rightarrow: \text{State} \rightarrow$

Int, які мають параметр \Rightarrow

$\Rightarrow x \in \text{Var}$:

- $\Rightarrow x(n) = [xan]$;
 \Rightarrow
- $x \Rightarrow (st) = st(x)$.

Тут і в подальшому вважаємо, що $n \in \text{Int}$, $st \in \text{State}$. Перша функція іменує іменем x число n , створюючи стан $[xan]$, друга бере значення імені x у стані st . Сама формула ґрунтується на тому факті, що стани змінних можуть тлумачитись як функції виду $\text{Var} \rightarrow \text{Int}$. Функція розіменування є частковою. Вона не визначена, якщо x не має значення у стані st .

Наприклад:

- $\Rightarrow M(5) = [Ma5]$;
- $Y \Rightarrow ([Ma3, Xa4, Ya2]) = 2$;
- $Y \Rightarrow ([Ma3, Xa4])$ не визначене.

Крім того, введемо

- параметричну функцію-константу арифметичного типу
 $n : \text{State} \rightarrow \text{Int}$, таку що $n(st) = n$ ($n \in \text{Int}$);
- тотожну функцію $\text{id} : \text{State} \rightarrow \text{State}$, таку, що $\text{id}(st) = st$.

Отримали багатоосновну алгебру даних мови SIPL

$A_{\text{Int_Bool_State}} = \langle \text{Int}, \text{Bool}, \text{State};$

$\text{add}, \text{sub}, \text{mult}, \text{or}, \text{neg}, \text{eq}, \text{gr}, \Rightarrow x, x \Rightarrow, n, \text{id}, \nabla \rangle,$

яка подана на рис. 1.3.

Зауваження 1.4. Є ще одна основа - Var . Оскільки ніяких операцій на цій основі у мові SIPL не задано, тому цю основу до алгебри даних не включаємо, але імена з Var використовуємо як параметри операцій іменування та розіменування.

1.3.2. Функції

Аналіз алгебри даних показує, що в мові SIPL можна вирізнити два види функцій: 1) n -арні функції на базових типах даних, 2) функції над станами змінних. Другий вид функцій будемо називати номінативними функціями. Назва пояснюється тим, що вони задані на наборах іменованих даних (латинське *nomen* - ім'я).

Визначимо тепер наступні класи функцій, які будуть задіяні при визначенні семантики мови SIPL:

1. n -арні операції над базовими типами:

- $FNA = \text{Int}^n \rightarrow \text{Int}$ - n -арні арифметичні функції (операції);

n

- $FNB = \text{Bool}^n \rightarrow \text{Bool}$ - n -арні булеві функції (операції);

n

- $FNAB = \text{Int}^n \rightarrow \text{Bool}$ - n -арні функції (операції) порівняння.

2. Функції над станами змінних

- $FA = \text{State} \rightarrow \text{Int}$ - номінативні арифметичні функції;
- $FB = \text{State} \rightarrow \text{Bool}$ - номінативні предикати;
- $FS = \text{State} \rightarrow \text{State}$ - біномінативні функції-перетворювачі (трансформатори) станів.

Для операцій мови SIPL зазвичай $n=2$, а для булевої операції заперечення $n=1$.

or (\vee)

\forall

add, sub, mult (+, -, *)

eq, gr (=, >)

Bool

Int

$x \Rightarrow, n$

$\Rightarrow x$

∇

State

id

neg (\neg)

Рисунок 1.3. Алгебра даних мови SIPL

1.3.3. Композиції

Нагадаємо, що композиції формалізують методи побудови програм. Аналіз мови SIPL дає підстави говорити про те, що будуть вживатися композиції різних типів, а саме:

- композиції, які пов'язані з номінативними функціями та предикатами;
- композиції, які пов'язані з біномінативними функціями.

Перший клас композицій використовується для побудови семантики арифметичних виразів та умов, другий – операторів.

Цей клас композицій складається із композицій суперпозиції в n -арні функції, які задані на різних основах (класах функцій):

- суперпозиція номінативних арифметичних функцій в n -арну арифметичну функцію має тип $S_n : FNA \times FA \rightarrow FA$;
- суперпозиція номінативних арифметичних функцій в n -арну функцію

порівняння має тип $S : FNAB \times FA \rightarrow FB$;

- суперпозиція номінативних предикатів в n -арну булеву функцію має тип

$S : FNB \times FB \rightarrow FB$.

Зауваження 1.5. Суперпозиції різного типу позначаємо одним знаком.

Суперпозиція задається формулою

$$(S_n(f_1, g_1, \dots, g_n))(st) = f(g_1(st), \dots, g_n(st)),$$

де f – n -арна функція, g_1, \dots, g_n – номінативні функції відповідного типу.

Другий клас композицій складається з наступних композицій.

- Присвоєння $ASx : FA \rightarrow FS$ (x – параметр). Присвоєння задається формулою:

$$AS \quad (fa) (st) = st \quad \forall [x \quad fa \quad (st)].$$

2

- Послідовне виконання $\cdot : FS \rightarrow FS$

Послідовне виконання задається формулою:

$$(fs \cdot fs) (st) = fs (fs (st)).$$

- Умовний оператор (розгалуження): $IF: FB \times FS \rightarrow FS$. Задається формулою:

$$IF (fb, fs_1, fs_2) (st) = \begin{cases} fs_1 (st), & \text{якщо } fb(st) = \text{true}, \\ fs_2 (st), & \text{якщо } fb(st) = \text{false}. \end{cases}$$

- Цикл (ітерація з передумовою): $WH: FB \times FS \rightarrow FS$.

Задається рекурентно

(індуктивно) наступним чином:

$$WH(fb, fs) (st) = st_n,$$

де $st_0 = st, st_1 = fs (st_0), st_2 = fs (st_1), \dots, st_n = fs (st_{n-1})$,

причому

$$fb (st_0) = \text{true}, fb (st_1) = \text{true}, \dots, fb (st_{n-1}) = \text{true}, fb (st_n) = \text{false}.$$

)=false .

$$0 \quad 1 \quad \dots \quad n-1 \quad n$$

Важливо відзначити, що для циклу наведена послідовність визначається однозначно. Позначимо число n (кількість ітерацій циклу) як $NumItWH((fb, fs), st)$. Однозначність визначення n дозволяє розглядати $NumItWH((fb, fs), st)$ як тернарне відображення, яке залежить від fb, fs, st . Якщо цикл не завершується, то $NumItWH((fb, fs), st)$ вважається невизначеним. Це відображення буде використовуватись в індуктивних доведеннях властивостей циклу.

1.3.4. Програмні алгебри

Побудовані композиції дозволяють стверджувати, що отримано алгебру функцій

(програмну алгебру)

$$A_Prog = \langle FNA, FNB, FNAB, FA, FB, FS; Sn$$

$$, AS, \cdot, \cdot \rangle,$$

$IF, WH, x \Rightarrow, id$.

Функції іменування $\Rightarrow x$ та накладання ∇ не включені до переліку операцій цієї алгебри, тому що вони представлені композицією присвоєння. Також не включаємо

функції-константи n , які мають специфічну природу. Разом із тим до переліку

композицій включено функції (нуль-арні композиції) розіменування та тотожна

функція. Це пояснюється тим, що вказані нуль-арні композиції мають не предметну (специфічну) природу, а логічну (загальнозначущу), що дозволяє розглядати їх саме

як загальні композиції (нехай і нуль-арні), а не як специфічні функції. Втім, цей

розподіл є досить умовним.

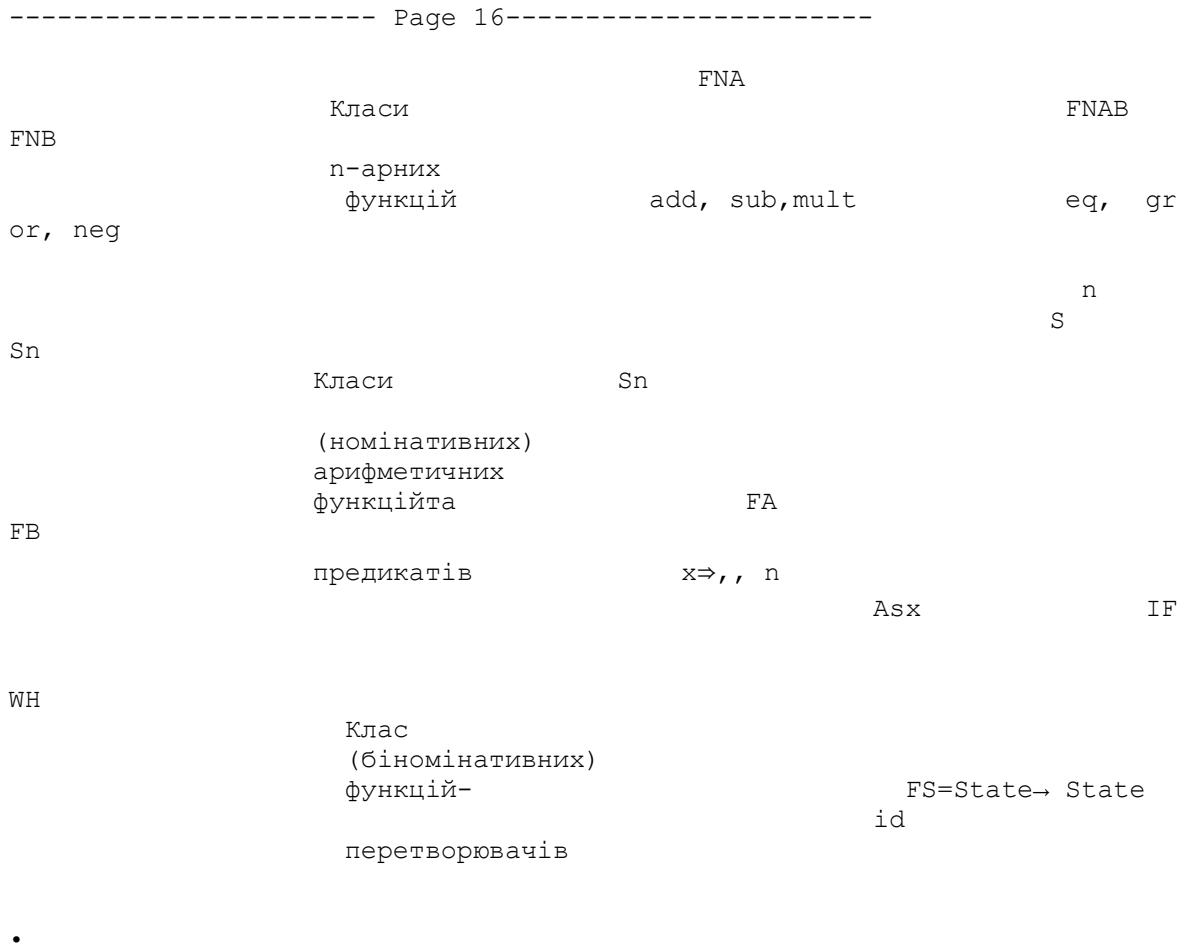


Рисунок 1.4. Алгебра функцій (програмна алгебра)

Наведена алгебра (рис. 1.4) в своїх основах містить багато «зайвих» функцій, які не можуть бути породжені в мові SIPL. Аналіз мови дозволяє стверджувати (цей факт буде доведено пізніше в теоремі 1.1), що функції, які задаються мовою SIPL, породжуються в алгебрі A_Prog з наступних базових функцій:

- $add, sub, mult \in FNA$;
- $or, neg \in FNB$;
- $eq, gr \in FNAB$;
- $n \in FA$ ($n \in Int$).

Підалгебру алгебри A_Prog , породжену наведеними базовими функціями, назвемо функціональною алгеброю мови SIPL і позначимо A_SIPL .

Зауважимо, що всі функції алгебри A_SIPL є однозначними (детермінованими) функціями. Це впливає з того, що всі базові функції є однозначними, а композиції зберігають цю властивість. Прохання до читача довести цю властивість самостійно. Формули для обчислення композицій та функцій алгебри A_Prog подамо у

наступній таблиці (тут f - n -арна функція, f_1, f_2, \dots, f_n - номінативні арифметичні функції, fb - номінативний предикат, f_1, f_2, f_3 - біномінативні функції, st - стан, n - число).

Таблиця 1.5

Композиція

Ім'я

Формула обчислення

формули

Суперпозиція
AF_S
$$(S(f_1, f_2, \dots, f_n))(st) = f_1(f_2(st), \dots, f_n(st))$$

Присвоєння
AF_AS
$$ASx(fa)(st) = st \nabla [xa \text{ fa}(st)]$$

Послідовне
AF_SEQ
$$fs_1 \cdot fs_2(st) = fs_2(fs_1(st))$$

виконання

Умовний оператор
true, AF_IF
$$\square fs_1(st), \text{ true}, fb(st) =$$

$$IF(fb, fs_1, fs_2)(st) = \square fs_2(st), \text{ false}, fb(st) =$$

false .

Цикл
AF_WH
$$WH(fb, fs)(st) = st, \text{ де}$$

$$st = st, st = fs_1(st), st = fs_2(st), \dots, st = fs_n(st)$$

$$fb(st_0) = true, fb(st_1) = true, \dots,$$

$$fb(st_{n-1}) = true, fb(st_n) = false .$$

Функція
AF_DNM розіменування
Тотожна функція
AF_ID
$$x \Rightarrow (st) = st(x)$$

$$id(st) = st$$

Зауваження 1.6. Наведені формули слід тлумачити з урахуванням частковості

функцій. А саме, якщо значення однієї з функцій, що фігурує у формулі, не є визначеним, то і результат не буде визначеним. Так, для формули $fs_1 \cdot fs_2(st) = fs_1(fs_2(st))$

1 2 2 1

вважаємо, що коли $fs_2(st)$ або $fs_1(fs_2(st))$ не визначені, то і результат не є визначеним.

1 2 1

Для роботи з частковими функціями використовують наступні позначення:

- $fs(st) \uparrow$ - значення fs на st не визначене,
- $fs(st) \downarrow$ - значення fs на st визначене,
- $fs(st) \downarrow = r$ - значення fs на st визначене і дорівнює r .

З урахуванням частковості, формулу для послідовного виконання можна записати у наступному вигляді:

$\square r$, якщо існує r' , $fs_1(st) \downarrow = r'$ & $fs_2(r') \downarrow = r$

$fs_1 \cdot fs_2(st) = \square r$

\square не визначено в інших випадках.

Формула для умовного оператора буде мати вигляд

$\square r$, якщо $fb(st) \downarrow = \text{true}$ & $fs_1(st) \downarrow = r$,
 $\square 1$
 $IF (fb, fs_1, fs_2)(st) = \square r$, якщо $fb(st) \downarrow = \text{false}$ & $fs_2(st) \downarrow = r$,
 $\square 2$

\square
 \square невизначено в інших випадках.

Інші формули можуть бути переписані аналогічним чином. Наведений вигляд формули зберігають і у випадку багатозначних (недетермінованих) функцій. Правда, тут такі функції розглядати не будемо.

Зазначимо, що наведені формули можна було б записувати із вживанням сильної рівності, яка задає невизначеність лівої частини при невизначеності правої. ■

Побудована алгебра A_SIPL дозволяє тепер формалізувати семантику програм мови $SIPL$, задаючи їх функціональними виразами (семантичними термами) алгебри A_SIPL .

1.3.5. Визначення семантичних термів

Досі ми не дуже чітко розрізняли функціональний вираз алгебри A_Prog та функцію, що задається цим виразом. Наприклад, запис $(f \cdot g) \cdot h$ можна тлумачити як функцію, і тоді її можна застосувати до стану st або як вираз, і тоді, наприклад, вивчати тотожність $(f \cdot g) \cdot h = f \cdot (g \cdot h)$. У математичній логіці таке розрізнення роблять явним, вважаючи, що $(f \cdot g) \cdot h$ є виразом (термом, формулою), а не функцією. Саму ж функцію, яка задається цим виразом, позначають, наприклад, $(f \cdot g) \cdot h$, де I -

I

інтерпретація символів f, g, h в алгебрі A_Prog .

Таке розрізнення можна було б зробити і для мови $SIPL$. У такому випадку для опису (дескрипції) функцій, які задаються програмами мови $SIPL$, можуть використовуватися функціональні вирази алгебри A_SIPL , які називаються термами цієї алгебри. Такі класи термів будуються індуктивно, аналогічно класам

синтаксичних категорій. Терми програмної алгебри будемо також називати семантичними термами. Ми зазвичай не будемо використовувати різні позначення для термів та функцій, сподіваючись, що читач із контексту зрозуміє, про яке поняття іде мова. Разом із тим слід пам'ятати, що це розрізнення є важливим у теорії програмування, яка чітко виокремлює синтаксис та семантику програм.

Зауважимо, що можна використовувати різні визначення термів алгебри, у тому числі

- індуктивне;
- рекурсивне;
- аналітичне;
- БНФ;
- графічне тощо.

Прохання до читача побудувати індуктивне визначення множини термів самостійно.

Позначимо множини термів, які задають арифметичні вирази, умови, оператори (та програми) мови SIPL відповідно як TFA, TFB, TFS.

1.3.6. Побудова семантичного терму програми

Програма мови SIPL може бути перетворена на семантичний терм (терм програмної алгебри), який задає семантику цієї програми (семантичну функцію

програми), перетвореннями такого типу:

- $sem_P: Prog \rightarrow TFS;$
- $sem_S: Stm \rightarrow TFS;$
- $sem_A: Aexp \rightarrow TFA;$
- $sem_B: Bexp \rightarrow TFB.$

Ці перетворення задаються рекурсивно (за структурою програми). Тому побудова семантичного терму залежить від вибору структури синтаксичного запису програми. Тут треба зважати на неоднозначність обраної нами граматики, що може призвести до різної семантики програм. Для досягнення однозначності треба користуватися пріоритетами операцій і типом їх асоціативності.

Таблиця 1.6

Номер

Правило заміни

правила

$sem_P: Prog \rightarrow TFS$ задається правилами:

$sem_P(\text{begin } S \text{ end}) = sem_S(S)$

NS_Prog

$sem_S: Stm \rightarrow TFS$ задається правилами:

$sem_S(x:=a) = AS \quad (sem_A(a))$

NS_Stm_As

$sem_S(S_1 ; S_2) = sem_S(S_1) \cdot sem_S(S_2)$

NS_Stm_Seq

1 2 1 2

$sem_S(\text{if } b \text{ then } S_1 \text{ else } S_2) = IF(sem_B(b), sem_S(S_1), sem_S(S_2))$

NS_Stm_If

1 2 1 2

```

sem_S(while b do S)= WH(sem_B(b), sem_S(S))
NS_Stm_Wh
sem_S(begin S end)=(sem_S(S))
NS_Stm_BE
sem_S(skip)=id
NS_Stm_skip

```

sem_A: Aexp → TFA задається правилами:

```

sem_A(n)= n
NS_A_Num
sem_A(x)=x⇒
NS_A_Var
sem_A(a +a )=S (add, sem_A(a ), sem_A(a ))
NS_A_Add
sem_A(a -a )=S (sub, sem_A(a ), sem_A(a ))
NS_A_Sub
sem_A(a *a )=S (mult, sem_A(a ), sem_A(a ))
NS_A_Mult

```

18

----- Page 19-----

```

sem_A((a))=sem_A(a)
NS_A_Par

```

sem_B: Bexp → TFB задається правилами:

```

sem_B(a =a )=S (eq, sem_A(a ), sem_A(a ))
NS_B_eq
sem_B(a >a )=S (gr, sem_A(a ), sem_A(a ))
NS_B_gr
sem_B(b ∨∨b )=S (or, sem_B(b ), sem_B(b ))
NS_B_or
sem_B(¬b)=S1(neg, sem_B(b))
NS_B_neg
sem_B((b))= sem_B(b)
NS_B_Par

```

Наведені правила слід розглядати як загальні правила, які в логіці називають

схемами правил. Щоб із загального правила (метаправила) отримати конкретне правило (об'єктне правило), слід замість синтаксичних метасимволів, таких як a , b , S , P , n , x , підставити конкретні синтаксичні елементи (записи), наприклад, замість a підставити $N-M$, замість b - $M > N$ і т.д. Далі ліва частина конкретного правила замінюється на його праву частину і т.д.

Приклад 1.2 Побудуємо семантичний терм виразу $X+Y*Z$. Побудова терму полягає в обчисленні значення $sem_A(X+Y*Z)$. Щоб зробити перший крок такого обчислення, треба віднайти правило, ліва частина якого буде збігатися із записом $sem_A(X+Y*Z)$

при відповідній конкретизації такого правила. Цей процес називається уніфікацією

двох записів (термів). У нашому випадку можлива уніфікація з лівою частиною

правил NS_A_Add та NS_A_Mult . У першому випадку уніфікація $sem_A(X+Y*Z)$ та

$sem_A(a + a)$ можлива при заміні a на X та a на $Y*Z$, а в другому уніфікація

$1 \quad 2 \quad 1 \quad 2$

$sem_A(X+Y*Z)$ та $sem_A(a * a)$ можлива при заміні a на $X+Y$ та a на Z . Наведені

$1 \quad 2 \quad 1 \quad 2$

2

заміни (підстановки) називаються уніфікаторами і зазвичай позначаються $[a / X, a$

$1 \quad 2$

$/Y*Z]$ та $[a / X+Y, a / Z]$ або $[a aX, a aY*Z]$ та $[a aX+Y, a aZ]$.

Зазначимо, що друга

$1 \quad 2 \quad 1 \quad 2 \quad 1 \quad 2$

уніфікація порушує пріоритет операцій, тому розглядаємо лише першу уніфікацію. Застосування першого уніфікатора до правила NS_A_Add призводить до наступного конкретного (об'єктного) правила:

$$NS_A_Add': sem_A(X+Y*Z) = S^{2}(add, sem_A(X), sem_A(Y*Z)).$$

Застосування цього правила дозволяє перетворити запис $sem_A(X+Y*Z)$ на запис

$S^{2}(add, sem_A(X), sem_A(Y*Z))$. Цей запис містить два підзаписи $(sem_A(X)$ та $sem_A(Y*Z))$, до яких можна застосувати перетворення.

Підзапис $sem_A(X)$

уніфікується з лівою частиною правила NS_A_Var за допомогою уніфікатора $[x/X]$, а

підзапис $sem_A(Y*Z)$ - з NS_A_Mult за допомогою уніфікатора $[a / Y, a / Z]$.

$1 \quad 2$

Застосування цих уніфікаторів призводить до двох нових конкретних правил:

$$NS_A_Var': sem_A(X) = X \Rightarrow \quad \text{та}$$

$$NS_A_Mult': sem_A(Y*Z) = S2(mult, sem_A(Y), sem_A(Z)).$$

Застосовуючи ці правила до виразу $S(\text{add}, \text{sem}_A(X), \text{sem}_A(Y*Z))$ отримаємо

$S(\text{add}, X \Rightarrow, S(\text{mult}, \text{sem}_A(Y), \text{sem}_A(Z)))$. Залишилося конкретизувати правило NS_A_Var , щоб отримати остаточний результат

$$\text{sem}_A(X+Y*Z) = S(\text{add}, X \Rightarrow, S(\text{mult}, Y \Rightarrow, Z \Rightarrow))$$

■

Отже, процес побудови семантичного терму програми полягає в послідовному перетворенні запису, для якого будується семантичний терм. Ці перетворення вимагають наступних дій:

- вибір загального правила, яке можна застосувати до підзапису поточного виразу з урахуванням пріоритету операцій;
- знаходження уніфікатора лівої частини правила з обраним під записом;
- отримання конкретного правила застосуванням уніфікатора до обох частин загального правила;

19

----- Page 20 -----

- заміна у поточному записі лівої частини конкретного правила на його частину.

Більш формально процес перетворень такого типу сформульований у теорії переписуючих правил.

Приклад 1.3. Побудувати семантичний терм програми GCD. Побудову будемо робити згідно з вищенаведеними правилами. Деталі не вказуємо.

$$\begin{aligned} \text{sem}_P(\text{GCD}) &= \text{sem}_P(\text{begin} \\ &\quad \text{while } \neg M=N \text{ do} \\ &\quad \quad \text{if } M>N \text{ then } M:=M-N \text{ else } N:=N-M \\ &\quad \text{end}) = \\ &= \text{sem}_S(\text{while } \neg M=N \text{ do if } M>N \text{ then } M:=M-N \text{ else } N:=N-M) = \\ &= \text{WH}(\text{sem}_B(\neg M=N), \text{sem}_S(\text{if } M>N \text{ then } M:=M-N \text{ else } N:=N-M)) = \\ &= \text{WH}(S(\text{neg}, \text{sem}_A(M=N), \\ &\quad \quad \quad \text{IF}(\text{sem}_B(M>N), \text{sem}_S(M:=M-N), \text{sem}_S(N:=N-M)))) = \\ &= \text{WH}(S(\text{neg}, S(\text{eq}, \text{sem}_A(M), \text{sem}_A(N)), \\ &\quad \quad \quad \text{IF}(\text{sem}_B(M>N), \text{sem}_S(M:=M-N), \text{sem}_S(N:=N-M)))) = \\ &= \text{WH}(S(\text{neg}, S(\text{eq}, M \Rightarrow, N \Rightarrow), \text{IF}(S(\text{gr}, M \Rightarrow, N \Rightarrow), \\ &\quad \quad \quad \text{AS}(\text{sem}_A(M-N), \text{AS}(\text{sem}_A(N-M)))))) = \\ &= \text{WH}(S(\text{neg}, S(\text{eq}, M \Rightarrow, N \Rightarrow), \text{IF}(S(\text{gr}, M \Rightarrow, N \Rightarrow), \\ &\quad \quad \quad M \quad N \quad 2 \end{aligned}$$

$$\begin{aligned}
& AS \quad (S \text{ (sub, sem_A(M), sem_A(N))), \\
& \quad N \quad 2 \\
& AS \quad (S \text{ (sub, sem_A(N), sem_A(M))})) = \\
& \quad 1 \quad 2 \\
& =WH(S \text{ (neg, S \text{ (eq, M}\Rightarrow, N\Rightarrow)}), \\
& \quad 2 \\
& \quad IF(S \text{ (gr, M}\Rightarrow, N\Rightarrow), \\
& \quad \quad M \quad 2 \\
& \quad \quad AS \quad (S \text{ (sub, M}\Rightarrow, N\Rightarrow)), \\
& \quad \quad N \quad 2 \\
& \quad \quad AS \quad (S \text{ (sub, N}\Rightarrow, M\Rightarrow))))
\end{aligned}$$

■

Повернемося тепер до доведення того факту, що семантика довільної програми мови SIPL задається термом алгебри A_S IPL . Дійсно, аналізуючи таблицю 1.6, бачимо, що там фігурують лише базові функції алгебри A_S IPL . Більш строгі доведення можна отримати індукцією за структурою програми SIPL. Таким чином, є справедливим наступне твердження.

Теорема 1.1. Для довільної програми мови SIPL її семантична функція задається термом алгебри A_S IPL.

Зауваження 1.7. Відображення побудови семантичного терму можна розглядати і в іншому аспекті – алгебраїчному. У цьому випадку програма розглядається як терм синтаксичної алгебри, який відображається в семантичну алгебру. Це відображення буде гомоморфізмом синтаксичної алгебри в семантичну. Доведення цього факту впливає з аналізу правил, заданих у таблиці 1.6. Прохання до читача довести цей факт самостійно.

1.3.7. Обчислення значень семантичних термів

Семантичні терми програми є точно (формально) заданими об'єктами, які формалізують семантику програм у термінах відповідних семантичних алгебр. Такі алгебри й терми і є головними об'єктами дослідження в нашому курсі. Ми далі будемо вивчати різні властивості таких алгебр та відповідних термів. Зараз розглянемо найпростішу властивість термів, зважаючи на те, що вони задають деякі функції (тобто повертаємось знову до синкретичного тлумачення терму як функції). Ця властивість називається аплікацією і полягає в застосуванні функції, що задається термом, до певних вхідних даних. Аплікація є аналогом (абстракцією) тестування програм.

Приклад 1.4. Обчислимо значення семантичного терму програми GCD на вхідному даному [M a8, N a16].

формули таблиці 1.6, є загальними правилами обчислень. Щоб їх застосовувати, треба робити конкретизацію загальних правил подібно до того, як описано в прикладі 1.3. Отже, завдання полягає в отриманні значення наступної аплікації:

$$\text{WH}(S(\text{neg}, S(\text{eq}, M \Rightarrow, N \Rightarrow)), \text{IF}(S(\text{gr}, M \Rightarrow, N \Rightarrow), M \Rightarrow, N \Rightarrow), \text{AS}(S(\text{sub}, M \Rightarrow, N \Rightarrow), \text{AS}(S(\text{sub}, N \Rightarrow, M \Rightarrow))))([M\ a8, N\ a16])).$$

Правила для обчислення циклу AF_WH говорять про необхідність поступового обчислення станів st, st, ... та перевірки відповідних умов. Отримуємо уніфікатор

$$[\text{fb} / S1(\text{neg}, S(\text{eq}, M \Rightarrow, N \Rightarrow)), \text{fs} / \text{IF}(S(\text{gr}, M \Rightarrow, N \Rightarrow), \text{AS}(S(\text{sub}, M \Rightarrow, N \Rightarrow), \text{AS}(S(\text{sub}, N \Rightarrow, M \Rightarrow))), \text{st} / [M\ a8, N\ a16], \text{st}0 / [M\ a8, N\ a16])].$$

Переходимо до обчислення fb(st), яке конкретизується аплікацією

$$S(\text{neg}, S(\text{eq}, M \Rightarrow, N \Rightarrow))([Ma8, Na16]).$$

Ця аплікація обчислюється згідно із загальним правилом AF_S для обчислення суперпозиції. Після відповідної уніфікації та конкретизації спочатку отримуємо, що

$$S(\text{neg}, S(\text{eq}, M \Rightarrow, N \Rightarrow))([Ma8, Na16]) = \text{neg}(S(\text{eq}, M \Rightarrow, N \Rightarrow))([Ma8, Na16]).$$

Застосовуючи правило обчислення суперпозиції ще раз та правила обчислення функції розіменування, отримуємо, що

$$S(\text{eq}, M \Rightarrow, N \Rightarrow)([Ma8, Na16]) = \text{eq}(M \Rightarrow([Ma8, Na16]), N \Rightarrow([Ma8, Na16])) = \text{eq}(8, 16) = \text{false}.$$

Остаточно маємо, що

$$S(\text{neg}, S(\text{eq}, M \Rightarrow, N \Rightarrow))([Ma8, Na16]) = \text{neg}(\text{false}) = \text{true}.$$

Правила обчислення для циклу говорять про необхідність обчислення за правилом

st = fs(st), що конкретизується наступним чином:

$$\text{IF}(S(\text{gr}, M \Rightarrow, N \Rightarrow), \text{AS}(S(\text{sub}, M \Rightarrow, N \Rightarrow), N \Rightarrow, N \Rightarrow))$$

AS (S (sub, N⇒, M⇒))([Ma8, Na16]).

Обчислення цієї аплікації полягає у застосуванні правила AF_IF для обчислення умовного оператора. Спочатку обчислюємо умову

2

S (gr, M⇒, N⇒)([Ma8, Na16])= gr(M⇒([Ma8, Na16]), N⇒([Ma8, Na16]))= gr(8, 16)= false .

N

2

При хибній умові за правилом AF_IF обчислюється AS (S (sub, N⇒, M⇒))([Ma8, Na16]). Отримуємо за правилом AF_AS

N 2

2

AS (S (sub, N⇒, M⇒))([Ma8, Na16])= [Ma8, Na16]∇[NaS (sub, N⇒, M⇒)([Ma8, Na16])] = [Ma8, Na16]∇[Nasub(N⇒([Ma8, Na16]), M⇒([Ma8, Na16]))] = [Ma8, Na16]∇[Nasub(16,8)]= [Ma8, Na16]∇[Na8] = [Ma8, Na8].

Отже, st конкретизується як [Ma8, Na8]. Тепер за правилом AF_WH обчислюємо

1

умову циклу на отриманому стані:

S1 2

(neg, S (eq, M⇒, N⇒))([Ma8, Na8])=false .

Таким чином, остаточний стан є [Ma8, Na8]. ■

1.3.8. Загальна схема формалізації мови SIPL

Підсумуємо схему формалізації, яка була застосована для мови SIPL.

Зазначимо,

що в першому наближенні мова L задається як трійка виду (Synt, Sem, interpretation),

де Synt - опис синтаксичного аспекту (опис текстів програм), Sem - опис семантичного аспекту (опис смислу програм),

interpretation: Synt→Sem -

інтерпретація програм, яка кожній програмі співставляє її смисл (значення).

Інтерпретацію також називають денотацією.

Спочатку ми мали початковий опис мови (Synt , Sem , interpretation), який

0

0

0

складався з формального подання синтаксису Synt0 у вигляді ВНФ та неформального опису семантики Sem . Інтерпретація interpretation : Synt →Sem також була

неформальною.

----- Page 22-----

Для формалізації семантики було вибрано формалізм функціональних алгебр. Цей формалізм також можна трактувати як певну мову, синтаксис якої задається термами алгебри Synt, семантика Sem - функціями з носія алгебри, а інтерпретація

1 1

interpretation : Synt → Sem є просто інтерпретацією термів в алгебрі.

1 1 1

Визначення формальної семантики надає можливість надати формальне визначення мови SIPL наступним чином:

- синтаксис Synt задається БНФ;

0

- семантика Sem задається функціями;

1

- інтерпретація interpretation : Synt → Sem є добутком відображень sem_P

SIPL 0 1

та interpretation, тобто

1

interpretation = sem_P • interpretation .

SIPL

1

(Тут множення • тлумачиться як композиція довільних відображень.)

Таким чином, мова SIPL уточнюється трійкою

(Synt, Sem, sem_P • interpretation).

0 1

1

Кожен компонент цієї трійки визначений формально. Вказана схема формалізації наведена на рис. 1.5.

	Початковий	Перший
формальний опис	(неформальний) опис мови	
класу функцій:	(класу функцій):	
(композиційний опис)	синтаксис:	
Синтаксис:		sem_P
формальний		
терми алгеб-		(формальне)
ри Synt1	(БНФ) Synt0	
		визначення

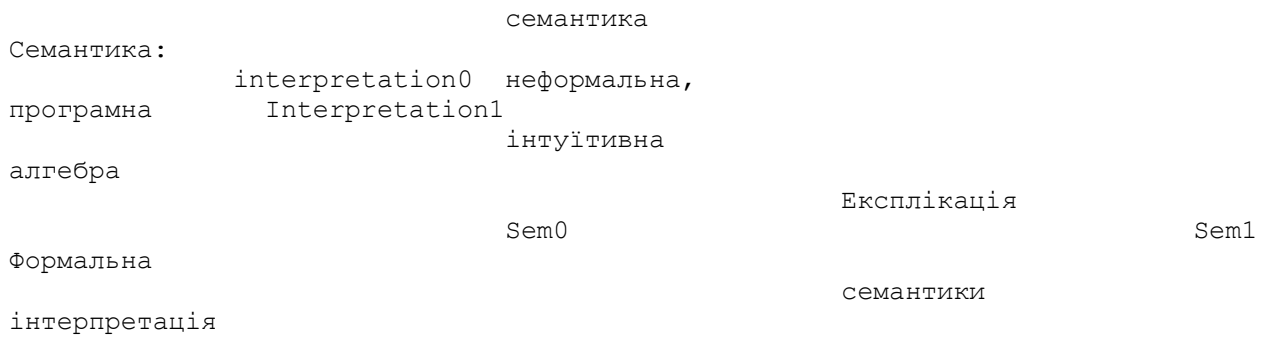


Рисунок 1.5 Схема визначення композиційної семантики

У цій схемі похідну стрілку Sem → Sem можна тлумачити як уточнення 0 1

(експлікацію) семантики.

Така схема буде використана далі для введення аплікативної аксіоматичної семантики.

1.4. Властивості програмної алгебри

Побудована програмна алгебра дозволяє сформулювати властивості програм, досліджуючи властивості функцій цієї алгебри, заданих її термами (функціональними виразами). Доведемо декілька таких властивостей.

Зауваження 1.8. У програмній алгебрі рівність розглядається як рівність функцій.

Лема 1.1. Доведемо властивість асоціативності послідовного виконання, тобто

справедливість наступної тотожності ($f, g, h \in FS$):

$$(f \cdot g) \cdot h = f \cdot (g \cdot h).$$

Доведення. Оскільки в лемі формулюється рівність функцій, то слід довести, що

на одному і тому ж даному обидві функції мають бути 1) одночасно невизначеними або 2) одночасно визначеними і в цьому випадку давати однакові результати

(сильна рівність функціональних виразів). Використовуючи позначення $fs(st)_{\downarrow}$

(функція fs визначена на st) та $fs(st)_{\uparrow}$ (функція fs невизначена на st) наведемо

формулювання рівності можна задати наступним чином:

$$fs_1 = fs_2 \text{ тоді і тільки тоді, коли для довільного стану } st \in State$$

$$1) ((fs_1(st)_{\uparrow} \& fs_2(st)_{\uparrow}) \text{ або}$$

$$2) (fs_1(st)_{\downarrow} \& fs_2(st)_{\downarrow} \& fs_1(st) = fs_2(st)).$$

Стосовно леми це означає:

$(f \cdot g) \cdot h = f \cdot (g \cdot h)$) тоді і тільки тоді, коли для довільного стану $st \in State$

- 1) $((f \cdot g) \cdot h)(st) \uparrow \& (f \cdot (g \cdot h))(st) \uparrow$ або
- 2) $((f \cdot g) \cdot h)(st) \downarrow \& (f \cdot (g \cdot h))(st) \downarrow \& ((f \cdot g) \cdot h)(st) = (f \cdot (g \cdot h))(st)$

Перейдемо до доведення. Беремо довільний стан $st \in State$. Спочатку доведемо, що якщо $((f \cdot g) \cdot h)(st) \uparrow$, то $(f \cdot (g \cdot h))(st) \uparrow$. Дійсно, для $((f \cdot g) \cdot h)(st)$ маємо формулу $h(g(f(st)))$. Тому значення $((f \cdot g) \cdot h)(st)$ буде не визначеним, якщо або $f(st)$ не визначене, або $g(f(st))$ не визначене, або $h(g(f(st)))$ не визначене. Але в кожному з цих трьох випадків буде невизначеним і значення $(f \cdot (g \cdot h))(st)$. Має місце і зворотне. Отже, якщо одне із значень $((f \cdot g) \cdot h)(st)$ або $(f \cdot (g \cdot h))(st)$ не визначене, то і друге значення також не визначене, тому формула $((f \cdot g) \cdot h)(st) \uparrow \& (f \cdot (g \cdot h))(st) \uparrow$ буде істинною. Зауважимо, що звідси

1 2

впливає, що якщо значення одного із функціональних виразів є визначеним, то є визначеним і значення іншого функціонального виразу.

Отже, коли одне із значень буде визначеним, то тоді визначені обидва значення і вони будуть рівними, бо $((f \cdot g) \cdot h)(st) = h(g(f(st)))$ та $(f \cdot (g \cdot h))(st) = h(g(f(st)))$.

Лема доведена. ■

У подальшому не будемо детально аналізувати випадки невизначеності функцій, сподіваючись, що читач сам зможе такі ситуації проаналізувати.

Зауваження 1.9. Рівність функцій можна доводити як рівність їх графіків. Але це вимагає переозначення композицій із функціональних у теоретико-множинні терміни. Тут цього робити не будемо.

Далі доведемо наступне твердження, яке часто буде використовуватися для доведення коректності програм із циклами.

Лема 1.2 (про властивості циклу). Для довільних функцій $fs \in FS$ і $fb \in FB$ та

довільного стану $st \in State$ мають місце наступні властивості:

- $WH(fb, fs) = IF(fb, fs \cdot WH(fb, fs), id)$.
- $NumItWH((fb, fs), st) = 0$ тоді і тільки тоді, коли $fb(st) \downarrow = false$ (це також означає, що $WH(fb, fs)(st) = st$).
- Якщо $NumItWH((fb, fs), st) > 0$, то
 - $fb(st) \downarrow = true$,
 - $WH(fb, fs)(st) = WH(fb, fs)(fs(st))$ та
 - $NumItWH((fb, fs), fs(st)) = NumItWH((fb, fs), st) - 1$.

Доведення. Спочатку доведемо першу властивість, що задає певну тотожність.

Беремо довільний стан $st \in State$. Припустимо, що $WH(fb, fs)(st)$ визначено. Можливі два варіанта:

- 1) $fb(st) = false$;
- 2) $fb(st) = true$.

У першому випадку тіло циклу не виконується ($NumItWH((fb, fs), st) = 0$), тому $WH(fb, fs)(st) = st$. При обчисленні $IF(fb, fs \cdot WH(fb, fs), id)(st)$ потрібно обчислити $id(st)$, що дає також значення st . Тому лема для цього випадку справедлива.

У другому випадку визначеність $WH(fb, fs)(st)$ означає, що є послідовність станів та

значень предиката (тому $\text{NumItWH}((fb, fs), st) > 0$), яка задовольняє наступним умовам:

$st_0 = st, st_1 = fs(st_0), st_2 = fs(st_1), \dots, st_n = fs(st_{n-1})$, причому

$fb(st_0) = true, fb(st_1) = true, \dots, fb(st_{n-1}) = true, fb(st_n) = false$.

Розглянемо, яке значення має $\text{WH}(fb, fs)(st)$. Зрозуміло, що послідовність обчислень

1

повторює вищеведену послідовність, але початковим станом є st . Тому

1

$\text{WH}(fb, fs)(st) = st$. Звідси також випливає, що $\text{NumItWH}((fb, fs), st) = n - 1$.

1

Подивимось тепер на обчислення функціонального виразу (праву частину тотожності)

$\text{IF}(fb, fs \cdot \text{WH}(fb, fs), id)(st)$.

Оскільки $fb(st) = true$, то

23

----- Page 24 -----

$\text{IF}(fb, fs \cdot \text{WH}(fb, fs), id)(st) = (fs \cdot \text{WH}(fb, fs))(st) = \text{WH}(fb, fs)(fs(st)) = \text{WH}(fb, fs)(st) = st$.

1 n

Таким чином, і в цьому випадку лема справедлива.

Аналогічно доводиться, що якщо визначена ліва частина тотожності, то буде визначена і права частина з тим самим результатом. Тотожність доведена. З цього доведення випливають і інші дві властивості, сформульовані в лемі ■

Зауваження 1.10. Тотожність $\text{WH}(fb, fs) = \text{IF}(fb, fs \cdot \text{WH}(fb, fs), id)$ стверджує, що $\text{WH}(fb, fs)$ є розв'язком відносно X (можливо, одним із розв'язків) функціонального рівняння

$X = \text{IF}(fb, fs \cdot X, id)$.

Дослідження рівнянь такого типу буде проведено у розділі, присвяченому рекурсії.

■ Лемі 1.1 та 1.2 характеризували властивості побудованої алгебри функцій A_Prog .

Таких властивостей є досить багато. Їх ідентифікація (формулювання) є важливою проблемою. Такі тотожності дозволяють робити еквівалентні перетворення програм з метою доведення їх властивостей або проведення оптимізації, трансляції, інтерпретації тощо.

Перейдемо тепер до визначення підалгебр алгебри A_Prog . Серед різних можливих підалгебр виділимо підалгебру, яка індукована аналізом відношення розширення (збагачення) станів новими змінними з їх значеннями. Наприклад, знаючи, що $\text{sem}_P(\text{GCD})([Ma8, Na16]) = [Ma8, Na8]$, запитаємо про значення $\text{sem}_P(\text{GCD})([Ma8,$

Na16, La9]), тобто про значення функції $\text{sem_P}(\text{GCD})$ на новому стані, який має нову змінну L. Більшість програмістів погодиться з тим, що результуючий стан буде просто розширенням попереднього результуючого стану цією новою змінною, тобто, що $\text{sem_P}(\text{GCD})([\text{Ma8}, \text{Na16}, \text{La9}]) = [\text{Ma8}, \text{Na8}, \text{La9}]$. Інакше кажучи, функція

$\text{sem_P}(\text{GCD})$ є монотонною щодо відношення розширення станів \subseteq .

Монотонність функцій визначається наступним чином.

Функція $f \in \text{FS}$ називається монотонною, якщо з того, що $f(s) \downarrow = str$ та $s \subseteq s'$,

впливає, що $f(s') \downarrow = str'$ та $str \subseteq str'$. Підклас монотонних функцій позначимо MFS .

Така властивість функцій використовується програмістами при збільшенні пам'яті комп'ютера, бо програми на збільшеній пам'яті будуть працювати так само (в будь-якому разі ми на це сподіваємось), як і на меншій пам'яті.

А чи є подібна властивість для функцій, породжених арифметичними виразами та умовами мови SIPL? Виявляється, при розширенні станів значення арифметичних функцій та предикатів не змінюється. Така властивість називається еквітонністю ("екві" означає "рівний"). Точне визначення наступне.

Функція $f \in \text{FA}$ називається еквітонною, якщо з того, що $f(s) \downarrow = r$ та $s \subseteq s'$ впливає, що $f(s') \downarrow = r$. Підклас еквітонних функцій позначимо EFA , підклас еквітонних предикатів - EFB .

Як бачимо, поняття монотонної та еквітонної функції є дуже важливими, тому подивимося на поведінку таких функцій стосовно композицій мови SIPL.

Виявляється (і зараз це буде доведено), що введені класи утворюють еквітонно-монотонну підалгебру

$$\begin{aligned} \text{A_EM_Prog} = \langle & \text{FNA}, \text{FNB}, \text{FNAB}, \text{EFA}, \text{EFB}, \text{MFS}; \\ & \text{Sn} \quad \quad \quad \text{x} \\ & , \text{AS} , \cdot , \text{IF}, \text{WH}, \text{x} \Rightarrow , \text{id} \rangle \end{aligned}$$

алгебри функцій A_Prog .

Для цього потрібно довести, що класи EFA , EFB , MFS замкнені відносно заданих на них композицій.

Спочатку доведемо, що клас EFA замкнений відносно композицій алгебри A_Prog , заданих на EFA , тобто що він є замкненим відносно композиції суперпозиції S_n та

функції розіменування. Нехай f - n-арна функція з FNA , $g_1, \dots, g_n \in \text{EFA}$. Треба довести,

1

n

що $S_n(f, g_1, \dots, g_n) \in \text{EFA}$. Беремо довільні s та s' , такі, що $s \subseteq s'$. Вважаємо також, що

$f(s) \downarrow = r$. Доведемо, що $(S_n(f, g_1, \dots, g_n))(s') \downarrow = r$. Дійсно, враховуючи, що

$g_1(s') = g_1(s), \dots, g_n(s') = g_n(s)$, маємо

n

$$(S(f_1, g_1, \dots, g_n))(st) = f_1(g_1(st), \dots, g_n(st)) = r.$$

Неважко також переконалися, що функція розіменування є еквітонною функцією з класу EFA.

Аналогічно доводиться, що і клас еквітонних предикатів EFB замкнений відносно відповідних композицій суперпозиції.

Залишилося довести, що клас монотонних функцій MFS замкнений відносно композицій присвоєння, послідовного виконання, умовного оператора, циклу та тотожної функції. Розглянемо всі ці випадки (для довільних st та st' , таких, що $st \subseteq st'$).

x

x

1. Присвоєння. Нехай $fa \in EFA$ та $AS(fa)(st) \downarrow = str$. За визначенням $AS(fa)(st) = st$

x

$\nabla[xa fa(st)]$. Обчислимо $AS(fa)(st')$, враховуючи, що для $fa \in EFA$ $fa(st') = fa(st)$. Маємо

$$ASx(fa)(st') = st' \nabla[xa fa(st')] = st' \nabla[xa fa(st)].$$

Порівнюючи $st \nabla[xa fa(st)]$ та $st' \nabla[xa fa(st)]$, можна стверджувати, що $st \nabla[xa fa(st)] \subseteq st' \nabla[xa fa(st)]$, оскільки значення змінної x у цих станах є однакові, а $st \subseteq st'$.

2. Послідовне виконання. Нехай $fs_1, fs_2 \in MFS$ та $fs_1 \cdot fs_2(st) \downarrow = str$. За визначенням,

$$fs_1 \cdot fs_2(st) = fs_1(fs_2(st)).$$

Обчислимо $fs_1 \cdot fs_2(st')$, враховуючи, що $fs_1, fs_2 \in MFS$. Маємо

$$fs_1 \cdot fs_2(st') = fs_1(fs_2(st')).$$

Оскільки $fs_1(st) \subseteq fs_1(st')$, то і $fs_1(fs_2(st)) \subseteq fs_1(fs_2(st'))$, що і треба

$$fs_1 \cdot fs_2(st) \subseteq fs_1 \cdot fs_2(st').$$

було довести.

3. Умовний оператор. Нехай $fb \in EFB, fs_1, fs_2 \in MFS$ та $IF(fb, fs_1, fs_2)(st) \downarrow = str$. За

$$IF(fb, fs_1, fs_2)(st) = \begin{cases} fs_1(st) & \text{якщо } fb(st) \\ fs_2(st) & \text{інакше} \end{cases}$$

$$IF(fb, fs_1, fs_2)(st') = \begin{cases} fs_1(st') & \text{якщо } fb(st') \\ fs_2(st') & \text{інакше} \end{cases}$$

визначенням, $IF(fb, fs_1, fs_2)(st)$ дорівнює $fs_1(st)$, якщо $fb(st)=true$, або $fs_2(st)$, якщо $fb(st)=false$.

Обчислимо $IF(fb, fs_1, fs_2)(st')$, враховуючи, що $fb \in EFB$ (тобто $fb(st')=fb(st)$),

а $fs_1, fs_2 \in MFS$. Тому, якщо $fb(st')=true$, то $IF(fb, fs_1, fs_2)(st')=fs_1(st')$, і оскільки $fs_1(st) \subseteq fs_1(st')$,

то $IF(fb, fs_1, fs_2)(st) \subseteq IF(fb, fs_1, fs_2)(st')$. Аналогічно для випадку $fb(st')=false$

отримуємо $IF(fb, fs_1, fs_2)(st) \subseteq IF(fb, fs_1, fs_2)(st')$.

4. Цикл. Доведення впливає з того, що послідовності станів при обчисленні $WH(fb, fs)(st')$ відповідають послідовності станів при обчисленні $WH(fb, fs)(st)$, причому увесь час має місце співвідношення $st_i \subseteq st'_i$ ($i=0, \dots, n$).

5. Тотожна функція є монотонною з класу MFS. Таким чином, доведено наступну лему.

Лема 1.3. Еквітонно-монотонна алгебра A_{EM_Prog} є під алгеброю алгебри функцій A_{Prog} .

Доведена лема дозволяє дати більш точний опис функцій, породжуваних мовою SIPL. Для цього слід показати, що алгебра A_{SIPL} є підалгеброю алгебри A_{EM_Prog} .

Слід переконатися, що функції-константи n є еквітонними функціями (що очевидно). Отже, справедливо наступне:

- якщо $a \in A_{Exp}$, то $sem_A(a) \in EFA$;
- якщо $b \in B_{Exp}$, то $sem_B(b) \in EFB$;
- якщо $S \in S_{tm}$, то $sem_S(S) \in MFS$;
- якщо $P \in Prog$, то $sem_P(P) \in MFS$.

Отримані твердження можна сформулювати у вигляді наступної теореми.

Теорема 1.2. Вирази та умови мови SIPL породжують еквітонні функції, а програми та оператори – монотонні функції.

З теореми випливає, що якщо програма мови SIPL завершується з якимись результатами на певному стані, то вона завершиться і на розширеному стані, причому однакові змінні в обох результуючих станах будуть мати однакові результати.

Доведемо тепер коректність програми GCD. Позначимо функцію взяття найбільшого спільного дільника двох чисел як gcd .

Відзначимо, що оскільки у нас поки що є лише одна - композиційна семантика, то коректність програми доводиться відносно цієї семантики. Тому програмістські терміни слід тлумачити саме у термінах композиційної семантики. Так, завершувальність програми означає, що функція, яка задає семантику програми, визначена на відповідному стані, ітерація циклу - обчислення наступного члена послідовності станів, і так далі.

25

----- Page 26-----

Теорема 1.3 (про часткову коректність програми GCD). Нехай стан st такий, що

$M \Rightarrow (st) = m, \quad N \Rightarrow (st) = n \quad (m, n > 0).$ Тоді, якщо $sem_P(GCD)(st) \downarrow = str,$ то $M \Rightarrow (str) = N \Rightarrow (str) = gcd(m, n).$

Зауваження 1.11. Згідно з теоремою 1.2 можна взяти стан, який має лише дві змінні - M та N . У цьому випадку теорему можна сформулювати більш просто: якщо $sem_P(GCD)([Mam, Nan]) \downarrow = [Mar1, Nar2],$ то $r1 = r2 = gcd(m, n).$

Доведення

Побудуємо семантичний терм функції $sem_P(GCD).$

Маємо

$sem_P(GCD) = sem_P(begin\ while\ \neg M = N\ do\ if\ M > N\ then\ M := M - N\ else\ N := N - M\ end) =$

$$= WH(S\ (neg,\ S\ (eq,\ M \Rightarrow,\ N \Rightarrow)),$$

$$2$$

$$IF(S\ (gr,\ M \Rightarrow,\ N \Rightarrow),$$

$$M\ 2$$

$$AS\ (S\ (sub,\ M \Rightarrow,\ N \Rightarrow)),$$

$$ASN\ 2$$

$$(S\ (sub,\ N \Rightarrow,\ M \Rightarrow))$$

$$)$$

$$)$$

Таким чином, семантика програми задається семантичним термом з композицією циклу у якості головної операції. Для спрощення позначимо

$$p_g = S\ (neg,\ S\ (eq,\ M \Rightarrow,\ N \Rightarrow)),$$

$$2$$

$$f_g = IF(S\ (gr,\ M \Rightarrow,\ N \Rightarrow), AS\ (S\ (sub,\ M \Rightarrow,\ N \Rightarrow)), AS\ (S\ (sub,\ N \Rightarrow,$$

$$M \Rightarrow))).$$

У цьому випадку $sem_P(GCD) = WH(p_g, f_g).$

Оскільки значення $WH(p_g, f_g)(st)$ визначене, то $NumItWH((p_g, f_g), st) \geq 0.$ Нехай $NumItWH((p_g, f_g), st) = k.$ Теорему будемо доводити індукцією за k (тобто за кількістю ітерацій циклу). Індуктивна гіпотеза фактично повторює формулювання теореми:

якщо $m > 0, n > 0, M \Rightarrow(st) = m, N \Rightarrow(st) = n, WH(p_g, f_g)(st) \downarrow = str$, то
 $M \Rightarrow(str) = N \Rightarrow(str) = \gcd(m, n)$.

База індукції. Нехай $k=0$. Згідно з лемою 1.2, $p_g(st) \downarrow = false$ та $WH(p_g, f_g)(st) = st$.

Що означає умова $p_g(st) \downarrow = false$? Щоб це з'ясувати, проведемо наступні

обчислення: $p_g(st) = S(neg, S(eq, M \Rightarrow, N \Rightarrow))(st) = ntg(eq(M \Rightarrow(st), N \Rightarrow(st))) = neg(eq(m, n)) = false$. Це означає, що $m = n$, тому $M \Rightarrow(st) = N \Rightarrow(st) = \gcd(m, n)$.

Теорема виконується для цього випадку.

Крок індукції. Нехай теорема справедлива для всіх станів st таких, що $NumItWH(p_g, f_g, st) = k$ (обчислення $WH(p_g, f_g)(st)$ вимагає k застосувань функції f_g на відповідних станах). Доведемо, що тоді теорема буде справедлива для станів st , на яких обчислення $WH(p_g, f_g)(st)$ вимагає $k+1$ кроків. В цьому випадку (згідно з лемою 1.2) це означає, що $p_g(st) \downarrow = true$ (тобто $m \neq n$), $WH(p_g, f_g)(st) = WH(p_g, f_g)(f_g(st))$ та $NumItWH(p_g, f_g, f_g(st)) = k$.

2 2 M
N
Проведемо обчислення $f_g(st) = IF(S(gr, M \Rightarrow, N \Rightarrow), AS(S(sub, M \Rightarrow, N \Rightarrow)), AS(S(sub, N \Rightarrow, M \Rightarrow)))(st)$.

Маємо два випадки:

М 2 2
1) $S(gr, M \Rightarrow, N \Rightarrow)(st) \downarrow = true$ (це означає, що $m > n$). Тоді $f_g(st) = AS(S(sub, M \Rightarrow,$

$N \Rightarrow))(st) = st \nabla [Ma S(sub, M \Rightarrow, N \Rightarrow)(st)] = st \nabla [Ma sub(M \Rightarrow(st), N \Rightarrow(st))] = st \nabla [Ma sub(m, n)] = st \nabla [Ma m - n]$. (Для стану st з двома змінними можна записати,

що результат є $[Mam, Nan] \nabla [Ma m - n] = [Ma m - n, Nan]$.);

2) $S2(gr, M \Rightarrow, N \Rightarrow)(st) \downarrow = false$ (це означає, що $m \leq n$, але враховуючи, що $m \neq n$,

N 2
2
маємо $m < n$). Тоді $f_g(st) = AS(S(sub, N \Rightarrow, M \Rightarrow))(st) = st \nabla [Na S(sub, N \Rightarrow, M \Rightarrow)(st)] = st \nabla [Na sub(N \Rightarrow(st), M \Rightarrow(st))] = st \nabla [Na sub(n, m)] = st \nabla [Na n - m]$.

(Для стану st з двома змінними можна записати, що результат є $[Mam, Nan]$

$$\forall [Na \ n- \ m] = [Mam, N \ a \ n- \ m].)$$

З теорії чисел випливає, що при вказаних умовах $\gcd(m, n) = \gcd(m, m-n)$, якщо $m > n$, та $\gcd(m, n) = \gcd(n-m, n)$, якщо $m < n$.

26

----- Page 27 -----

Дійсно, не обмежуючи загальності, розглянемо випадок $m > n$. Тут M має значення $m-n$, а $N - n$. Покажемо, що $\gcd(m, n) = \gcd(m, m-n)$. Для цього слід показати, що множина спільних дільників m і n і множина спільних дільників $m-n$ і n співпадають. Нехай q - спільний дільник m і n , тобто $m = q * k$, $n = q * r$, для деяких натуральних чисел k, r . Тоді $m-n = q * k - q * r = q * (k-r)$, тобто q - спільний дільник $m-n$ і n . Аналогічним чином доводиться і зворотне твердження. Оскільки множини спільних дільників двох пар чисел співпадають, то і найбільший спільний дільник у них теж співпадає.

Отже, у стані $st = f_g(st)$ значення змінних M та N додатні (більше нуля) та

1

дорівнюють відповідно m та $m-n$, або $n-m$ та n . Таким чином, для обох випадків

маємо $M \Rightarrow (st) > 0, N \Rightarrow (st) > 0$, та $\gcd(M \Rightarrow (st), N \Rightarrow (st)) = \gcd(M \Rightarrow (st),$

$N \Rightarrow (st)) = \gcd(m, n)$.

1

1

1

1

Тепер можна скористуватися індуктивною гіпотезою для стану st , бо всі іі

1

засновки виконуються. Тому отримуємо, що $M \Rightarrow (str) = N \Rightarrow (str) = \gcd(M \Rightarrow (st), N \Rightarrow (st))$.

1

1

З урахуванням того, що $\gcd(M \Rightarrow (st), N \Rightarrow (st)) = \gcd(m, n)$, маємо, що

1

1

$M \Rightarrow (str) = N \Rightarrow (str) = \gcd(m, n)$.

Теорему доведено.

■

Відзначимо дві обставини, пов'язані з теоремою коректності.

По-перше, теорему доведено за умови додатності значень змінних M та N . А як буде працювати програма, коли ця умова не виконується? Коли значення змінних M та N не є додатними та рівні між собою (наприклад, значення змінних M та N

дорівнюють -5), то програма зупиняється, не змінюючи стан. Але наявне від'ємне число (або 0), що є значенням змінних, не може вважатися їх найбільшим спільним дільником. Коли ж значення змінних різні і хоча б одне із значень є від'ємним або 0 , то програма зациклюється. Отже, на правильний результат можна сподіватися лише для додатних значень змінних M та N .

По-друге, теорему коректності доведено за умови завершуваності програми. Така

коректність називається частковою коректністю. Повна (тотальна) коректність програми P для певного класу ST входних даних означає, що програма є частково коректною та завершується на всіх входних даних із цього класу. Якщо позначити формулу завершуваності програми P на стані $st \in ST$ як $termination(P)(st)$, а коректність - як $correctness(P)(st)$, то умова часткової коректності задається формулою

$$\forall st \in ST (termination(P)(st) \Rightarrow correctness(P)(st),$$

а тотальної коректності -

$$\forall st \in ST (termination(P)(st) \wedge correctness(P)(st).$$

Теорема 1.4 (про повну коректність програми GCD). Нехай стан st є такий, що

$M \Rightarrow (st) = m$, $N \Rightarrow (st) = n$ та $m, n > 0$. Тоді для деякого стану str маємо, що

$$sem_P(GCD)(st) \downarrow = str \text{ та } (M \Rightarrow (str)) = (N \Rightarrow (str)) = gcd(m, n).$$

Щоб отримати повну коректність програми для додатних значень змінних M та N , треба довести завершуваність програми (точніше, визначеність значення функції

$WH(p_g, f_g)(st)$). Це означає, що треба довести визначеність значення $NumItWH(p_g, f_g)(st)$, якщо $m, n > 0$.

Розглянемо послідовність станів змінних

$$st_0 = st, st_1 = fs(st_0), st_2 = fs(st_1), \dots$$

Оскільки при обчисленні нових станів задіяна лише функція віднімання, яка є всюди визначеною, то кожен елемент послідовності є визначеним, а сама послідовність є нескінченною.

Доведемо, що в цій послідовності є стан, у якому значення змінних M та N

співпадають. Доведення проведемо від супротивного. Нехай це не так, тобто нехай у

цій послідовності для кожного стану значення M та N різні. Розглянемо тепер

послідовність сум значень цих змінних, тобто послідовність

$$t_0 = M \Rightarrow (st_0) + N \Rightarrow (st_0), t_1 = M \Rightarrow (st_1) + N \Rightarrow (st_1), t_2 = M \Rightarrow (st_2) + N \Rightarrow (st_2), \dots$$

У початковому стані значення M та N додатні, тому $t_0 > 0$. За відсутності рівних

0

значень M та N додатними мають бути і всі інші числа t_1, t_2, t_3, \dots . Разом із тим

0

----- Page 28-----

послідовність є строго спадною, тобто $t_0 > t_1 > t_2 > \dots$, тому що нові значення M та N

$t_0 \ t_1 \ t_2$

отримуються відніманням меншого з чисел від більшого. За наведених умов послідовність t_0, t_1, t_2, \dots не може бути нескінченною, бо обмежена знизу числом 2,

$t_0 \ t_1 \ t_2$

що є найменшою сумою двох додатних чисел.

Отримане протиріччя говорить про те, що в послідовності станів змінних $st = st_0$,

st_0

$st_1 = fs(st_0)$, $st_2 = fs(st_1)$, ... є стан з однаковими значеннями M та N , на якому цикл

$t_1 \ t_0 \ t_1$

зупиняється, тобто значення $WH(p_g, f_g)(st)$ буде визначеним.

Це означає, що для додатних значень змінних M та N доведено повну коректність нашої програми.



Зауваження 1.12 Звернемо ще раз увагу на те, що при доведенні теореми ми

інколи застосовували програмістську термінологію, говорячи, наприклад, про ітерації циклу, про завершуваність програми, а не про обчислення послідовності

певних значень функцій, як того вимагає композиційна семантика. Зроблено це для того, щоб доведення краще відповідало програмістській інтуїції. Втім, сподіваємося

на те, що читач зможе розпізнати та трансформувати програмістські терміни у точні

математичні формулювання.

Варто підкреслити ще одну обставину: а саме, що без наявності формальної

семантики та синтаксису ми не могли б навіть говорити про доведення різних

властивостей програм та, зокрема, їх коректності. На інтуїтивному рівні розуміння

програм може бути лише інтуїтивне обґрунтування властивостей програм.

Висновки

У цьому розділі був розглянутий простий приклад програми в мові SIPL. Ця мова є

одним із варіантів простих мов подібного типу, наприклад, мови WHILE [14].

Фактично такі мови є мовами структурного програмування. Однак тут була дана

композиційна семантика мови SIPL, яка базується на запропонованому В.Н. Редьком композиційному програмуванні [10]. Таким чином, основні результати цього розділу полягають у наступному:

1. Дано неформальний та формальний описи мови SIPL.

2. Для формального визначення синтаксису мови запропоновано її ВНФ

та

розглянуто її властивості (дерева виводу, однозначність та неоднозначність БНФ).

3. Для формального визначення семантики мови побудовано алгебру даних мови SIPL та низку функціональних алгебр. Операціями цих алгебр є композиції, що

формалізують засоби конструювання програм.

4. Визначено відображення побудови семантичного терму програми.

5. Досліджено низку тотожностей в алгебрі програм.

6. Продемонстрована можливість доведення часткової та повної коректності на підставі введених формалізацій.

Разом із тим виникає низка питань. Можливо, всі введені поняття і методи

спрацьовують лише на простій мові для більш складних мов це не спрацює? Тобто чи є введені поняття необхідними та суттєвими, чи, може, вони випадкові? Які

поняття ще слід увести, щоб працювати з більш потужними мовами програмування?

Питання такого типу є загальнометодологічними питаннями. Їх розглянемо в наступному розділі.

Контрольні питання

1. Які недоліки неформального опису мов програмування?
2. Яким чином задається синтаксис мови SIPL?
3. Що є деревом синтаксичного виводу програми?
4. Коли програма є синтаксично правильною?
5. Що таке неоднозначна БНФ?
6. Для чого вводиться пріоритет операцій?
7. Які пріоритети введено для операцій мови SIPL?

28

----- Page 29-----

8. Які синтаксичні категорії та метазмінні введені в мові SIPL?
9. Які типи даних використовуються у мові SIPL?
10. Які алгебри даних пов'язані з мовою SIPL?
11. Як визначаються операції іменування та розіменування?
12. Які класи функцій використовуються для формалізації семантики мови SIPL?
13. Які типи номінативних функцій використовуються для формалізації семантики мови SIPL?
14. Як визначається суперпозиція в n-арну функцію?
15. Як визначається композиція присвоєння?
16. Як визначається композиція послідовного виконання?
17. Як визначається композиція розгалуження?
18. Як визначається композиція циклу?
19. Які програмні алгебри пов'язані з мовою SIPL?
20. За якими критеріями в перелік композицій включають і функції?
21. Як визначається підалгебра алгебри A_Prog , породжена мовою SIPL?
22. Як частковість функцій впливає на визначення композицій?
23. Як визначається семантичний терм програми?
24. Як будується семантичний терм програми?
25. Які властивості виконуються для програмних алгебр?
26. Як визначаються монотонні функції?
27. Як визначаються еквітонні функції?
28. Які визначають програми мови SIPL?
29. Що таке часткова коректність програм?
30. Що таке повна коректність програм?

Вправи

1. Довести однозначність синтаксичного аналізу при використанні правил пріоритету та асоціативності операцій.
2. Побудувати рекурентні співвідношення для визначення синтаксичних категорій.
3. Навести приклади неоднозначного аналізу програм мови SIPL.
4. Навести індуктивне визначення множин термів програмної алгебри.
5. Побудувати програми на мові SIPL для наступних задач (x , y , n – додатні цілі числа):

- обчислення $x*y$, використовуючи функції $+$, $-$ (та не використовуючи функції $*$);
- обчислення $n!$;
- обчислення $x-y$, використовуючи функцію віднімання одиниці (-1) ;
- перевірки парності числа n ;
- обчислення $x \text{ div } y$, використовуючи функції $+$, $-$;
 y
- обчислення $x \text{ mod } y$, використовуючи функції $*$, $+$, $-$;
- обчислення $[\lg n]$, використовуючи функції div , mod , $+$, $-$;
- обчислення $x \text{ mod } y$, використовуючи функції $+$, $-$;
 x
- обчислення 3^x , використовуючи функції $*$, $+$, $-$.

6. Перевірити синтаксичну правильність програм, створених у вправі 5, побудувавши їх дерева синтаксичного виводу. Побудувати семантичні терми цих програм та застосувати їх до певних вхідних даних.

7. Довести часткову та повну коректність програм, створених у вправі 5.

8. Довести, що програми мови SIPL задають однозначні (детерміновані) функції.

9. Довести лему про співпадіння для програм мови SIPL, а саме, визначити за програмою мови SIPL змінні, від яких залежить програма (індукцією за структурою програми), та довести, що значення програми будуть однаковими на станах з однаковими значеннями таких змінних.

10. Побудувати консервативні розширення мови SIPL. Тут неформально вважаємо, що розширення SIPL_C є консервативним розширенням SIPL, якщо для кожної програми з мови SIPL_C можна побудувати еквівалентну програму з мови SIPL. Включити до консервативного розширення n -арні функції div , mod , abs ; композиції repeat_until , if_then , цикли for тощо. Надати формалізацію розширеної мови.

2. Розвиток основних понять програмування

Постійне розширення сфери застосування обчислювальної техніки і необхідність побудови все більш складних, але надійних програмних систем змушують дослідників знову і знову звертатися до визначення основних понять

програмування. Незважаючи на різноманіття існуючих моделей програм і парадигм програмування, не можна не визнати тієї обставини, що розробка загальноприйнятної теорії програмування ще не завершена. Водночас зрозуміло, що центральною складовою такої теорії повинні бути строгі уточнення (експлікації) понять програми і програмування. Одним із підходів до початкових уточнень цих понять є композиційно-номінативний підхід, який може розглядатися як подальший розвиток композиційного програмування.

Цей підхід базується на наступних трьох основних принципах: розвитку, композиційності і номінативності. Принцип розвитку (від абстрактного до конкретного) говорить про поступове уточнення поняття програми, починаючи з найбільш абстрактних і продовжуючи більш конкретними і багатими уточненнями. Принцип композиційності трактує програми як функції, що будуються з інших функцій за допомогою спеціальних операцій, які називаються композиціями. Принцип номінативності говорить про необхідність використовувати відношення іменування для роботи з програмами і даними.

Ідеї розвитку, композиційності і номінативності добре відомі і простежуються з глибокої давнини. Категорія розвитку є однією з центральних категорій у такому філософському напрямку, як діалектика, серед представників якої виділимо Геракліта Ефеського та Г.В.Ф. Гегеля. Композиційні властивості мовних виразів були чітко сформульовані Г. Фреге та Р. Карнапом. Принцип композиційності був основним у композиційному програмуванні, запропонованому В.Н. Редьком. Різним аспектам номінативності (іменування) присвятили свої роботи Аристотель, Гоббс, Джон Стюарт Мілль, Джордж Буль, Пірс, Фреге, Рассел, Кріпке і багато інших.

Перш ніж переходити до детального розгляду основних понять програмування, проаналізуємо визначення поняття «програма», наведені у словниках.

2.1. Аналіз словникових визначень поняття програми

Почнемо з визначень поняття програми, які наводяться у словниках і енциклопедіях. Великий тлумачний словник сучасної української мови, Меріам-

Убстер Електронний словник та інші словники дають наступне трактування слова «програма».

Етимологія: від грецького *prographēin* – попередній запис (*pro* – перед + *graphēin* –

писати). Пізніше латинське *programmata* : письмове повідомлення. Значення :

- наперед продуманий план якої-небудь діяльності, роботи тощо;
- план для програмування деякого механізму (як комп'ютер);
- послідовність кодів інструкцій, що можна ввести в механізм

(наприклад, комп'ютер).

Аналізуючи ці та інші визначення терміна «програма», дійдемо висновку, що визначення використовують такі терміни, як «план», «послідовність», «дія», «інструкція», «програмування», «комп'ютер» і т.д. Подивимося, наприклад, на

визначення першого терміна в цьому списку. «План» – це:

1

<http://www.slovnyk.net/>

2 <http://www.m-w.com/cgi-bin/dictionary>.

3 Серед різних значень слова 'програма' виділимо лише ті, які близькі до предмету нашого розгляду.

- креслення або діаграма,
- метод досягнення кінцевої мети,
- упорядкована класифікація частин проекту або мети,
- детальна програма дій.

Бачимо, що визначення терміна «план», які нас цікавлять, звертаються знову до слова «програма». Звідси можна зробити два висновки:

1) словникові визначення поняття програми призводять до хибного кола, бо

ніякого глибокого розвитку цього поняття не відбувається, а є лише досить проста тавтологія, коли фактичного розкриття терміна немає, а стверджується, що програма – це те саме, що і план, а план є програмою;

2) поняття програми є дуже широким поняттям, яке не визначається як конкретизація деякого іншого поняття (не визначається за схемою родовидових понять Аристотеля).

Подібна ситуація буде і з іншими термінами з нашого списку. Правда, критикуючи словникові визначення за неповноту, зауважимо, що вони надають початкове уявлення про значення різних термінів. Тому треба погодитися зі словами М.Т. Рильського:

Не бійтесь заглядати у словник:
це пишний яр, а не сумне провалля.

Таким чином, термін 'програма' є надзвичайно загальним, багатоаспектним і не може бути простим способом визначений через інші терміни. Але як же тоді визначати такі загальні терміни?

Сформулювавши питання таким чином, ми фактично ставимо запитання про методи наукового пізнання і тим самим полишаємо область програмування і переходимо у більш загальну сферу, що яка є розділом філософії і яка називається гносеологією. Гносеологія (також вживається термін епістемологія) визначається як наука про взаємини суб'єкта та об'єкта, про закони, форми і методи пізнання, а також відношення знання до дійсності, критерії його істинності і достовірності. Тут суб'єкт та об'єкт – філософські категорії: під суб'єктом розуміється той, хто досліджує об'єкт.

2.2. Розвиток поняття програми з гносеологічної точки зору

Насамперед необхідно обґрунтувати, що поняття програми дійсно є складним, багатоаспектним поняттям і не зводиться тривіальним чином до більш простих понять. Це стає зрозумілим, якщо згадати історію розвитку програм, що починалася з досить простих обчислювальних програм для перших комп'ютерів і потім поступово перетворилася на потужну галузь виробництва, яка залучає величезні людські і фінансові ресурси. Складність поняття програми зрозуміла і зі словникових визначень. Уточнення таких складних, різнобічних та багатоаспектних понять має відбуватися згідно із загальними законами гносеології. Сформулюємо це твердження у вигляді наступного принципу.

Принцип гносеологічності. Уточнення основних понять програмування здійснюється відповідно до загальних законів (принципів) гносеології, застосування яких має програмологічну спрямованість.

Наведений принцип має два аспекти. Перший стосується доцільності використання загальних законів (принципів) гносеології для уточнення основних понять програмування. Другий аспект говорить про те, що ці закони слід конкретизувати, орієнтуючись на предметну область, що досліджується, та на теорію, що будується, тобто на програмування та на теорію програмування (програмологію). Тут під програмологією розуміється наука про програми та процеси програмування.

Гносеологія як наука про пізнання повинна, зокрема, надати нам загальні закони уточнення понять конкретних предметних областей. Складність використання гносеології з цією метою полягає в тому, що, незважаючи на багатовікову історію її розвитку, все ж не створені загальноприйняті закони наукового пізнання. Існують різні школи і напрямки, що часто трактують одні і ті самі питання по-різному. Через це для дослідника конкретної предметної області залишаються три можливості:

- приєднатися до однієї зі шкіл гносеології;
- розробити власний гносеологічний підхід;
- ігнорувати загальногносеологічні проблеми.

Кожна з можливостей має свої переваги та недоліки. Розробити власний підхід дуже складно. Це призведе до значних витрат на чисто філософську діяльність, займе багато часу. Ігнорування гносеологічних проблем призведе до примітивних помилок, які вже давно зрозумілі філософам. Тут є повна аналогія з теорією дебютів у шахах. Розробити власний дебют дуже важко, бо історія шахів нараховує тисячоліття і в ній зроблено надзвичайно багато. Ігнорування теорії дебютів призведе до програшу вже на початку партії. Тому шахісти, як правило, використовують вже розроблені та досліджені дебюти. Так і ми спробуємо використати певний гносеологічний підхід. Але який? Звичайно той, що стоїть найближче до практики програмування.

Тому слід визначити ті особливості програмування, які повинні знайти своє обґрунтування в гносеологічному підході. Це, перш за все, програмування як поступове уточнення, програмування від абстрактного до конкретного. Це підтримується такими методами, як структурне, систематичне, об'єктно-орієнтоване програмування тощо. Виявляється, є гносеологічний підхід, який саме і проголошує метод розвитку (побудови від абстрактного до конкретного) своїм головним методом. Цей підхід називається діалектичною логікою.

Отже, в межах даної роботи будемо спиратися саме на цей напрямок гносеології, який був розвинутий Г.В.Ф. Гегелем та його послідовниками. Правда, для наших цілей немає потреби використовувати діалектичну логіку у всій її потужності. Будемо використовувати лише ті елементи діалектичної логіки, які необхідні для опису понять програмування.

Основною складовою діалектичної логіки є система взаємопов'язаних та взаємообумовлених категорій, які тут розглядаються як всезагальні ознаки предметів. Прикладами категорій є категорії абстрактне - конкретне, об'єктивне - суб'єктивне, якість - кількість - міра, елемент - частина - ціле, форма - зміст, явище - сутність, одиничне - особливе - всезагальне та багато інших.

Система категорій утворює каркас нашого мислення. Розбудова такої системи спирається на низку принципів.

Першим принципом є розвиток від абстрактного до конкретного: перехід від абстрактних, бідних за змістом визначень до конкретних, більш багатих за змістом. Другий принцип - це рух мислення від однієї категорії до протилежної за значенням з подальшим їх об'єднанням у третій категорії (тріадичний розвиток). Третій принцип фактично є певним уточненням перших двох принципів. Згідно з ним категорії членуються на три групи, що відповідають трьом етапам пізнання. Перша група - це зовнішні визначення предметів, друга - внутрішні визначення, третя - визначення самого поняття. На першому етапі предмет визначається з боку його зовнішності, на другому визначаються його внутрішня побудова, а на третьому поєднуються зовнішні визначення предмета з його внутрішніми визначеннями.

Наведені етапи дослідження предмета часто формулюють у вигляді наступної схеми: синкретичний етап (предмет не членується на складові, розглядаються зовнішні відношення), аналітичний етап (розглядається внутрішня структура та відношення між складовими) та синтетичний етап (об'єднуються зовнішні та внутрішні відношення та складові).

Сформульовані принципи вказують на методи розбудови системи категорій. Та основний смисл прихований у самих категоріях. Виділяють близько сотні категорій, які всі пов'язані між собою. Таких зв'язків - тисячі. Тому немає простого викладу системи категорій. Проводячи аналогію з математикою, можемо казати, що визначення системи категорій задається складною системою з сотень рівнянь над

33

----- Page 34-----

сотнями змінних. Разом із тим для визначення основних понять програмування ми і не будемо використовувати усю потужність системи філософських категорій, а

обмежимося лише тими моментами, які пов'язані з програмуванням. Цей висновок впливає і з принципу гносеологічності, але сформулюємо його як спеціальний принцип.

Принцип програмологічної проекції категорій: при визначенні основних понять програмування категорії розглядаються лише в тих аспектах, які відображають властивості програмування.

Зв'язки категорій часто формулюються як закони діалектики. Зупинимося більш докладно на законі заперечення заперечення. У діалектичній логіці заперечення розглядається як зміна (розвиток), що змінюючи річ (заперечуючи її), приводить до нової речі, яка зберігає в деякій іншій формі властивості початкової речі. Закон заперечення заперечення говорить про те, що, зміна заперечує змінюване, результат у свою чергу змінюється новим запереченням, що призводить нас не до простого повторення початкового, а до його нового рівня розвитку. Тим самим цей закон стверджує спадкоємність, зв'язок нового зі старим, повторювальність на вищій стадії розвитку деяких властивостей нижчої стадії.

Нарешті, надамо певні пояснення категорії абстрактне - конкретне. Інколи ці терміни тлумачаться як уявне (ідеальне) - реальне. Тут ми беремо інше значення цих термінів. Абстрактне (походить від латинського abstraction - відволікання, виділення) означає одностороннє, просте, нерозвинуте. Конкретне (від латинського concretus - зрощене) означає єдність різних сторін, багатостороннє. Тут важливо відзначити, що конкретність розглядається саме як єдність, як системність.

Процес опису деякого предмета (речі, об'єкта) фіксується за допомогою понять. Як було сказано, розвиток понять будемо здійснювати від абстрактного до конкретного. Починаємо з найбільш абстрактних (але істотних) понять. Зміст понять задається за допомогою суджень, що розкривають властивості предмета. Сукупність суджень про поняття на визначеному етапі дослідження будемо називати рівнем абстракції, на

якому розглядається поняття. На кожному рівні абстракції можна робити умовиводи, що формулюють властивості предмета дослідження. Крок конкретизації полягає в тому, що вводяться поняття і судження, які розкривають нові властивості предмета. На отриманому новому рівні абстракції робляться нові умовиводи і т.д.

Застосовуючи зазначені положення до програм, одержуємо наступний принцип.

Принцип розвитку (від абстрактного до конкретного). Програмні поняття уточнюються у процесі їх розвитку. Цей процес починається з найбільш абстрактних уявлень, які відображають загальні істотні властивості програм та програмування, потім поступово переходить до більш конкретних уявлень у їх єдності, які відображають специфічні властивості програм та програмування, і таким чином, нарешті, розкриває поняття програмування в їх багатстві та взаємозв'язках.

Слід відзначити, що наведений принцип (та й інші методологічні принципи та визначення) не слід тлумачити абсолютно. Зокрема, цей принцип безумовно слід розглядати в єдності з рухом пізнання від конкретного до абстрактного. Тому завжди треба брати до уваги відносність аспектів, переходи одних аспектів в інші та пам'ятати в цілому про наявність складних взаємовідношень між поняттями.

Сукупність таких відношень називають діалектикою понять.

Наведений принцип розвитку є дуже загальним, тому сформулюємо деякі наслідки, які розкривають його зміст. Насамперед відзначимо, що оскільки поняття будуть розглядатися на різних рівнях абстракції, то при їх вивченні варто чітко

- вказувати рівень абстракції, що розглядається (принцип фіксації рівня абстракції);
- при побудові умовиводів використовувати тільки такі властивості понять, що були явно визначені раніше (принцип достатньої підстави);
- не використовувати властивості, задані на інших рівнях і явно не визначені на рівні, що розглядається (принцип інкапсуляції рівнів абстракції);
- у процесі розвитку попередні властивості не відкидаються, а зберігаються на нових рівнях розвитку, можливо, в іншій формі (принцип кумулятивності розвитку).

Ще одним наслідком принципу розвитку є принцип генетичності, який стверджує, що структури програм є похідними від їх генетичних структур (структур породження).

Підставою вибору того чи іншого напрямку розвитку є принцип єдності логічного та історичного. Цей принцип може розглядатися як певний варіант відомого гносеологічного принципу єдності теорії та практики.

Розвиток понять часто відбувається згідно з тріадою розвитку: теза - антитеза - синтез.

Розглянемо простий приклад. Припустимо, що ми хочемо дати визначення терміна «теща» (теза). З цим терміном асоціюється інший термін - «зять» (антитеза). Наявність цих двох термінів (які протистоять один одному) ще не дає можливості безпосередньо визначити перший термін. Необхідно зробити ще один крок розвитку (синтез), що поєднає терміни «теща» та «зять». Це поєднання задається особою, що є дружиною зятя та дочкою тещі.

Наведений розвиток призвів до визначення терміна «теща», що приводиться у словниках: теща - мати дружини.

Теща (теза)

заперечення

Зять (антитеза)

Дружина
(синтез)

Рисунок 2.1. Тріада розвитку

Підсумовуючи наведені вище гносеологічні положення, можна говорити про наступне.

1. Предмети світу пов'язані між собою. Це емпіричний факт, що часто формулюється як принцип взаємозв'язку предметів (речей). Як наслідок отримуємо, що кожен предмет є єдністю багатьох сторін (аспектів).

2. Оскільки кожен предмет (крім дуже простих) має багато сторін, то неможливо визначити його одним визначенням.

3. Зважаючи на неможливість охопити всі сторони предмета одним визначенням, спочатку зупиняються лише на деяких аспектах (тобто предмет спочатку розглядається абстрактно).

4. Подальше визначення (уточнення) поняття предмета може відбуватися шляхом конкретизації обраних аспектів (як правило за родовидовою схемою Аристотеля) або (що є більш загальною схемою) за тріадою розвитку, коли далі обираються інші аспекти (тим самим вони є запереченням початковообраних аспектів) та які потім інтегруються з початковообраними аспектами.

5. Вибір аспектів визначається метою дослідження, але важливим є вибір суттєвих аспектів, що ведуть до визначення предмета по суті.

6. Цей вибір аспектів, як правило, відповідає наступній схемі: зовнішні аспекти - внутрішні аспекти - взаємозв'язок аспектів.

Наведені принципи мають застосовуватися як на найвищому рівні (філософському, методологічному), так і на інших рівнях розгляду - професійному (науковому) та формальному (математичному).

Озброївшись основними гносеологічними принципами, перейдемо до розгортання основних понять програмування на професійному (науковому) рівні.

2.3. Розвиток основних понять програмування

Для уточнення поняття програми будемо розглядати походження (генезис) тих понять, що виникають у процесі його розвитку. При цьому прагнемо, щоб нові поняття, які виникають, були дуже простими і не вимагали для свого розуміння багатого досвіду програмування. У цьому сенсі запропоноване уточнення (експлікація) поняття програми може розглядатися як початкова експлікація.

Звертаємо увагу на те, що термін «поняття» буде вживатися в наступних смислах:

1) в інтуїтивному, коли поняття розглядається у всьому багатстві його властивостей, 2) частково експлікованому, коли з поняттям пов'язуються тільки ті властивості, що визначені на фіксованому рівні абстракції. Будемо звертатися до інтуїтивного розуміння у тому випадку, коли необхідно ввести нову властивість, яка глибше розкриває зміст досліджуваного поняття. Частково експліковане розуміння використовується для вивчення властивостей поняття на фіксованому рівні абстракції. Сподіваємося, що з тексту завжди буде зрозуміло, про яке тлумачення йде мова.

Наша мета полягає в тому, щоб не просто ввести якісь визначення понять програмування, а продемонструвати їх зв'язки і те, як від одних понять ми мусимо переходити до інших.

Наприклад, ми хочемо визначити поняття університету. Перший аспект, який можна виділити (абстрагувати) в цьому понятті, - це те, що університет складається із студентів. Але це дуже абстрактна характеристика університету. Переходимо до (діалектичного) заперечення цього аспекту: університет - це викладачі. Але знову таке визначення неповне. Далі ідемо до синтезу: університет - це студенти та викладачі, пов'язані

навчальним процесом (лекції, семінари тощо). Це вже досить конкретне визначення. Але і така тріада є абстрактною. Під таке визначення університету потрапляють і дитячий садок, і школа. Тому треба вводити ще більш конкретні визначення, розкриваючи структуру університету з різних сторін (навчальні плани, факультети, приміщення тощо.) Тільки на цьому шляху поняття університету буде розкрито конкретно, у його складових та їх зв'язках, призначенні університету і т.д.

Така сама ситуація має місце і для понять програмування. Розглянемо ще один простий приклад. Припустимо, що містер NN хоче дізнатися про рух коштів на його банківському рахунку. Він зв'язується з банківським службовцем, для якого проблема містера NN перестає бути одиначною проблемою, а стає масовою: як за номером банківського рахунку з'ясувати стан рахунку. Для розв'язання цієї проблеми службовець використовує спеціальну програму. Але ця програма є однією з багатьох програм, таких як системи керування базами даних, операційні системи тощо, які розробляються і обслуговуються спеціалістами банку. Цей приклад показує як, починаючи з «маленької» індивідуальної проблеми, ми змушені переходити до все більш загальних проблем, які вимагають для свого розв'язання усе більш складних і розвинених програмних систем. Приблизно таким же способом будемо розвивати поняття програми, починаючи з найбільш простих абстрактних властивостей і продовжуючи більш спеціальними і багатими властивостями. Отже, процес розвитку веде нас від індивідуальних властивостей через особливі до (все)загальних. Як уже відзначалося раніше, у процесі уточнення програмних понять будемо часто використовувати тріаду розвитку: теза – антитеза – синтез.

Уточнення поняття програми природно починати, як говорили у давнину, *ab ovo* – із самого початку. Але що обрати за початок?

Можна починати від понять програми та програмування. Але наведені приклади говорять про те, що для уточнення цих понять усе одно необхідно вийти на всезагальні поняття. Такою всезагальністю є поняття людства (суспільства). З цього і почнемо розвиток понять програмування.

2.3.1 Початкова тріада понять програмування

При пізнанні сутності програм та програмування ми повинні перш за все виходити з єдності трьох визначень їх понять: одиначності, особливості та (все)загальності. Цим категоріям відповідають такі моменти розгляду досліджуваного предмета, як елемент, частина та ціле. Зрозуміло, що сфера програмування належить до такої цілісності, як людство (суспільство). Але це занадто багата цілісність. Вона містить такі особливості, як сфери освіти, охорони здоров'я, оборони тощо. Серед цих особливих сфер є і сфера практичної діяльності, яка пов'язана з програмами та програмуванням. Назвемо її сферою інформатизації. Це занадто широкий термін, але і більш вузький термін важко вказати. Тому згідно з принципом програмологічної проекції будемо як всезагальне для теорії програмування брати цю сферу.

Таким чином, сфера інформатизації відноситься до практики. Що ж стосується теорії, то будемо (як уже зазначалося) виокремлювати три рівні (рис. 2.2):

- філософський, на якому визначаються категорії,
- науковий, на якому визначаються (загально- та спеціально-) наукові поняття,
- формальний, на якому визначаються формальні поняття.

Рисунок 2.2. Сфера інформатизації як особлива сфера суспільства

Розвиток понять програмування почнемо з найвищого (філософського) рівня. На цьому рівні основним є категорії суб'єкта та об'єкта. Оскільки ми говоримо про орієнтацію на практику, то будемо розглядати категорію суб'єкта у праксеологічному (діяльнісному) аспекті. Діяльність суб'єкта має бути цілеспрямованою. Тобто, ціль є антитезою суб'єкта. Їх синтезом є категорія засіб. Таким чином, отримано першу тріаду: суб'єкт – ціль –

засіб. Розвиваючи далі цю триаду отримуємо ще дві категорії - використання засобів та створення засобів. Отримуємо пентаду: суб'єкт - ціль - засіб - використання засобів - створення засобів.

Переходимо тепер до наукового рівня розгляду. Тут найважливішим чинником буде поняття користувача (його елементами будуть конкретні користувачі). Саме це поняття - користувача - і є початковим пунктом розвитку. А що можна назвати його діалектичним запереченням? Користувачу (як і людині в цілому) протистоїть зовнішній світ, той світ, у якому людина існує, який вона сприймає, перш за все, як сукупність проблем, що потребують розв'язку. Тому можна вважати, що ціллю користувача є розв'язок певної проблеми. Згідно з триадою розвитку наступний крок приведе до поняття, яке є синтезом понять користувача і проблема і яке може також розглядатися як розвиток користувача. Це - розв'язок проблеми. Цей розв'язок, враховуючи принцип програмологічної проекції, слід конкретизувати стосовно області програмування. У цій області розв'язком вважається програма (яка повинна допомогти користувачу розв'язати його проблему). Точніше кажучи, нас цікавить програмний розв'язок. Цей розв'язок з'являється саме як програма, як план. Тут можна навести слова К.Маркса про те, що найгірший архітектор відрізняється від бджоли тим, що попередньо має ідеальний образ, план. Таким чином, отримали першу триаду основних понять програмування. Тут користувач - теза, проблема - антитеза, програма - синтез.

Ця триада вперше вводить термін «програма», і саме ця триада задає умови існування і ціль (мету) програм. Тому цю триаду доцільно назвати триадою цільового призначення програм. Мета програм полягає у тому, що вони є засобами розв'язку певних проблем користувача.

У цій триаді розвитку можна також говорити про аналогії між введеними поняттями і такими категоріями гносеології, як «суб'єкт - мета (ціль) - засіб». Інакше кажучи, можна говорити про те, що поняття користувача, проблеми та програми є програмологічними проекціями відповідно до категорій суб'єкта, цілі та засобу.

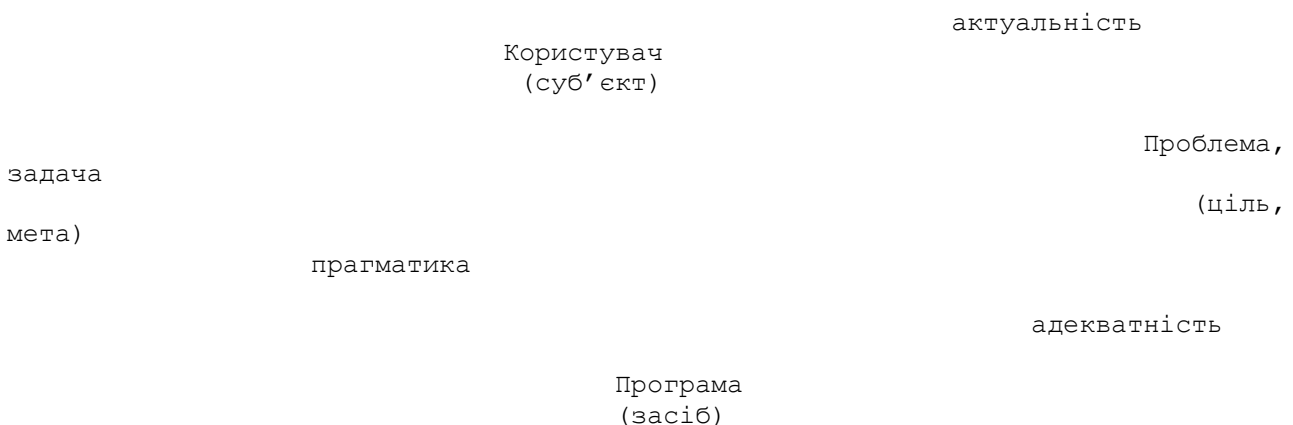


Рисунок 2.3. Триада цільового призначення програм

Відзначимо, що введенням понять користувача, проблеми та програми не вичерпується розвиток понять програмування. Дуже важливо, що введені поняття пов'язані низкою співвідношень. Дійсно, розглянемо користувача та його проблеми. Серед них він повинен у першу чергу вирішувати ті проблеми, які є важливими для нього. Використовуючи традиційний для наукових досліджень термін «актуальний», будемо говорити, що основним відношенням між користувачем та проблемами є відношення актуальності (проблеми для користувача).

Далі розглянемо, який зв'язок між проблемами та програмами. Якщо користувач обрав для розв'язку актуальну для нього проблему, то йому бажано мати програму, яка найбільш відповідним чином (найбільш адекватно) розв'язує цю проблему. Тому

поняття проблеми і програми пов'язані відношенням адекватності (відповідності, правильності програми для розв'язку проблеми).

Поняття користувача і проблеми пов'язує відношення прагматичності (від грецького прагма – діло, дія). Відмітимо, що назва цього відношення має і певну епістемологічну обґрунтованість: «користувач» походить від «користуватися», «діяти».

Сформульовані відношення відображено на рис. 2.3. Ці відношення є основними. Але крім них, є ще інші відношення, які ми зараз не розглядаємо як суттєві.

Тепер слід зробити наступний крок розвитку. Цей наступний крок полягає в розкритті відношення прагматичності.

2.3.2. Тріада прагматичності програм

Розглядаючи користувача як тезу, а програму – як антитезу, ми зараз повинні віднайти синтезуюче поняття, яке розкриває спосіб користування програмами.

Згідно з принципом програмологічної перспективи потрібно конкретизувати це поняття таким чином, щоб наблизитися до програмування (як програмування, орієнтованого на обчислювальну техніку). Ця конкретизація полягає у розкритті засобів реалізації цього відношення. Процес обчислення саме і є таким засобом. Це означає, що необхідно ввести поняття процесу виконання (за допомогою певного виконавця, наприклад комп'ютера). Враховуючи, що процес виконання відбувається, як правило, за допомогою комп'ютера, будемо такий процес також називати процесом обчислення програми.

Отже, використання програми полягає в першу чергу в її виконанні. З'являється нова тріада, яку назовемо тріадою прагматичності програм:

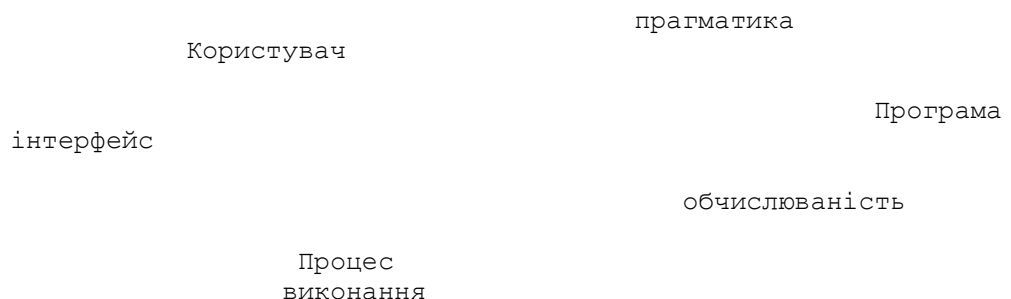


Рисунок 2.4. Тріада прагматичності програм

Основні відношення в цій тріаді наступні:

- прагматика (програма – користувач);
- обчислюваність (програма – процес виконання);
- інтерфейс (процес виконання – користувач).

Відзначимо, що хоча на перших етапах розвитку понять програмування введені відношення є абстрактними та бідними, при подальшій конкретизації вони розгортаються до дуже важливих та багатих відношень. Так, відношення обчислюваності є предметом теорії алгоритмів, якій присвячено чимало книжок. Інше відношення «користувач – процес обчислення». Його можна називати по-різному: (людино-машинний) інтерфейс, діалог. Це відношення деталізується як в монографіях, так і в державних стандартах. Можна виокремлювати різні рівні абстракції, пов'язані з цим відношенням. Наприклад:

1. Найбільш абстрактним буде вивчення відношення «користувач – процес обчислення» – в абстракції від програм та проблем.

2. Більш конкретним буде вивчення цього відношення з урахуванням особливостей програм.

3. Ще більша конкретність з'явиться при урахуванні особливостей програм та проблем тощо.

На кожному рівні абстракції є свої специфічні особливості відношення інтерфейсу. Наприклад, на першому рівні можна врахувати сприйняття користувачем кольорів монітору, швидкість набору символів на клавіатурі. На другому рівні можна врахувати кількість

інформації, що видається програмою, спосіб розміщення цієї інформації на моніторі тощо. На третьому рівні додатково враховуються особливості проблеми, скажімо, чи вимагає проблема реакції у реальному часі. Звичайно, є і багато інших конкретних рівнів.

2.3.3. Тріада основних понять програмування

Перейдемо до наступного кроку розвитку. Цей крок полягає в тому, щоб розкрити відношення адекватності. Це відношення розкривається у процесі створення, породження програм. Цей аспект називається генетичним аспектом програм. На цьому кроці з'являється поняття, що поєднує поняття програми та проблеми. Таким поняттям буде програмування (як процес), яке повинно пов'язати разом проблему і відповідну їй програму. Таким чином, побудовано нову тріаду розвитку: проблема (теза) – програма (антитеза) – процес програмування (синтез). Цю тріаду назвемо тріадою основних понять програмування, бо саме в ній уперше з'являється поняття програмування (тут терміни програмування та процес програмування розглядаємо як синоніми, проте будемо часто вживати саме термін «процес програмування», щоб підкреслити динамічний характер програмування). Крім того, термін «основні» тут використаний для того, щоб підкреслити, що суть програмування полягає саме у створенні необхідної єдності проблеми предметної області і програми, що використовується для розв'язання цієї проблеми.

Поняття основної тріади програмування пов'язані певними піввідношеннями (рис. 2.5). Так, поняття програмування та проблеми з'єднує відношення орієнтованості процесу програмування на розв'язок певної проблеми (проблемно-орієнтованість). Що ж стосується взаємодії понять програми та програмування, то тут виділимо два відношення:

- відношення генетичності програм (відношення «програма – процес програмування»);
- відношення експлікативності процесу програмування (відношення «процес програмування – програма»).

Для розкриття основної ідеї цих відношень розглянемо приклади.

Припустимо, що ви хочете купити мобільний телефон. Серед різних параметрів, які визначають ваш вибір, буде і параметр, пов'язаний із виробником телефону, тобто яка фірма і на якому заводі виробила телефон. Якщо це – бренд (на сучасному етапі – Nokia, Samsung, Motorola), то ви маєте більше довіри до якості телефону. Цей приклад демонструє важливість генетичного аспекту. Те саме стосується і генетичності програм: якщо програма (програмна система) розроблена відомою фірмою, яка використовує гарні технології розробки програм, то слід очікувати, що така програма буде більш надійною та ефективною, ніж програма, створена фірмою-одноденкою.

Відношення процесу програмування до програми буде конкретизовано як відношення уточнюваності (експлікативності процесу програмування).

Щоб зрозуміти, що означає це відношення, треба розглянути, як процес програмування пов'язаний із програмою. Спочатку про програму нічого не відомо (ні її розмір, ні кількість операторів і т.д.). Програма виступає як чорна скринька, наповнення якої з'явиться пізніше. На наступних кроках процесу програмування вимальовується деяка структура програми, її частини і т.д. Програма поступово уточнюється. Процес цих уточнень і задає відношення експлікативності.

Зробимо термінологічне зауваження: інколи вживають терміни «деталізація», «конкретизація», «декомпозиція». В англійській літературі часто вживається термін «refinement», одним із смислів якого є уточнення.

Проблема

проблемна
орієнтованість

Процес програмування

адекватність

експлікативність
та генетичність

Програма

Рисунок 2.5. Тріада основних понять програмування

Уведені відношення також пов'язані одне з одним. Так, можна говорити про те, що відношення адекватності програми проблемі є певним «добротком» відношень проблемно-орієнтованості та уточнюваності: чим вони кращі, тим кращою є адекватність програми. Тобто можна записати таку символічну (але не формальну) формулу:

$$\begin{aligned} &(\text{адекватність програми проблемі}) = \\ &= (\text{проблемно-орієнтованість процесу програмування}) * \\ &* (\text{експлікативність процесу програмування}) \end{aligned}$$

2.3.4. Пентада основних понять програмування

У попередніх підрозділах було побудовано три тріади розвитку, що в сукупності вводять п'ять основних понять програмування (на професійному рівні). Ці п'ять понять та їх співвідношення будемо називати пентадою основних понять програмування. Ця пентада утворена трьома обертами розвитку (конкретизації). Її розвиток почався з філософського рівня (рис. 2.6).

Пентада основних понять програмування виділяє основні аспекти понять програмування. Так, для поняття програми основними аспектами будуть

- адекватність;
- прагматичність;
- обчислюваність;
- генетичність.

Для процесу програмування основні аспекти це

- проблемна орієнтованість;
- експлікативність.

Зробимо також кілька методологічних зауважень стосовно побудованої пентади. Відзначимо, що для нас тут головним є самі поняття та їх відношення, а не порядок появи у процесі розвитку. Процес розвитку можна було б починати від процесу програмування та йти до програми, потім до проблеми, користувача та процесу виконання. Разом із тим, може з'явитися питання: чому другий оберт (користувач – програма – процес виконання) зроблено раніше, ніж третій (проблема – програма – процес програмування)? Обґрунтування задається принципом історичності. На початковому етапі використання програм велика вага приділялась ефективності використання машинного часу (бо був дуже дорогий), тобто проблемам виконання програм. Лише пізніше почали досліджуватися методи програмування.

У посібнику значну увагу приділено категоріям абстрактного та конкретного. Проілюструємо ці категорії на різних визначеннях поняття програмування, які індукуються введеною пентадою.

Рисунок 2.6. Схема розвитку основних понять програмування

Перше визначення: програмування є процес побудови програм. Це визначення поєднує лише два поняття нашої пентади та абстрагується від інших понять.

Друге визначення: програмування є процес побудови програм, що має на меті розв'язання певних проблем. Це визначення поєднує вже три поняття нашої пентади, але абстрагується від інших понять.

Третє визначення: програмування є процес побудови програм, що має на меті розв'язання певних проблем, актуальних для користувача. Це визначення поєднує чотири поняття нашої пентади, але абстрагується від інших понять.

Нарешті, четверте визначення: програмування є процес побудови програм, що має на меті розв'язання певних проблем, актуальних для користувача, шляхом їх виконання. Це визначення поєднує усі поняття нашої пентади.

Кожному наведеному визначенню відповідають певні методи програмування. Так, перше визначення фактично концентрується на методах написання синтаксично правильних програм. Тут можна навести аналогію з навчанням у школі: спочатку дітей вчать писати букви та слова (тобто тут концентруються на синтаксичному аспекті в абстракції від семантичного). Друге визначення вже апелює до предметної області, тому відповідні методи програмування мають враховувати семантику програм. Продовжуючи аналогію зі школою, можна сказати, що на цьому рівні діти повинні правильно викладати думки. Третє визначення апелює до користувачів та їх особливостей. На цьому рівні дітей у школі повинні навчати специфіці звертання та розмов з учителями, учнями, людьми похилого віку тощо. Нарешті, четверте визначення говорить про те, що методи програмування повинні враховувати особливості інтерфейсу та обчислюваності програм.

Таким чином, введена пентада дозволяє скласти певне уявлення про програмування. Та все ж уведено поняття є ще дуже абстрактними. Побудована пентада фактично відображає будь-який процес програмування, зокрема, процес підготовки програм для телебачення і т.д. Це пентада програмування взагалі. Наприклад, за цією пентадою ми можемо складати план роботи на наступний тиждень. Тому необхідно рухатися до більш конкретних понять, що розкривають процес програмування саме в професійному аспекті, у аспекті інформатизації.

Як же рухатися далі? Аналізуючи побудовану пентаду, можемо дійти висновку, що в них мова йде про зовнішні властивості програм та програмування. Згідно з принципами гносеології маємо тепер перейти до розкриття внутрішніх аспектів цих понять.

2.4. Розвиток основних програмних понять

Ми почнемо з розгляду внутрішніх аспектів програм. Відповідні поняття будемо називати програмними поняттями. Їх слід відрізнити від понять програмування, що пов'язані з процесом програмування, та які були розглянуті у попередньому розділі.

2.4.1. Тріада основних програмних понять

Почнемо подальший розвиток із конкретизації поняття програми. Вибір цього поняття (а не поняття користувача, проблеми, процесу виконання, процесу програмування) пов'язаний з тим, що це поняття є первісним щодо поняття процесу програмування, бо останнє «включає» в себе поняття проблеми і програми. Поняття програми також є первісним і для процесу виконання, що має спиратися на певне

розуміння програм. Що стосується понять проблеми та користувача, то вони є більш важливими, ніж поняття програми, є визначальними для нього, але їх розгляд буде виводити нас поза сферу інформатизації. Тому доцільно починати розкриття основних понять програмування з поняття програми, яке легше піддається уточненню, ніж інші поняття пентади.

Як же починати уточнення поняття програми? Є різні підходи до того, як це робити. Але ми почнемо з аналізу тієї триади, в якій уперше з'явилося поняття програми. Це триада «користувач – проблема – програма». З цієї триади ми бачимо, що поняття програми є діалектичним запереченням (антитезою) поняття проблеми, тобто програми не є проблемами, але зберігають у собі проблеми у якійсь іншій формі. Цей зв'язок понять проблеми та програми подано на рис. 2.7.

Яка ж саме властивість проблем знаходить своє відображення у програмах? Щоб сформулювати ту властивість проблем, яка робить можливим використання програм для їх розв'язання, подивимося на визначення проблеми. Філософські словники дають дуже гарне і просте визначення проблеми як єдності відомого та невідомого. З гносеологічної точки зору, тут мова йде про відому та невідому інформацію (знання).

Розглянемо питання про інформаційну властивість проблем більш детально. Сучасні комп'ютерні словники визначають інформацію як дані, інтерпретовані в

4 The Free On-line Dictionary of Computing. [Електронний ресурс].
– Домен доступу:

http://work.ucsd.edu:5141/cgi-bin/http_webster

предметній області. Вони також дають наступну низку термінів, пов'язану з інформацією:

дані – інформація – знання – мудрість.

Користувач

Проблема

Програма

Програма

запереченням

проблеми

Область
програм

Предметна
область

Малюнок 2.7. Програма як діалектичне заперечення проблеми

Тут дані тлумачаться як форма подання інформації, інформація – це дані, інтерпретовані у нашому світі, знання – істинна інформація, мудрість – уміння використовувати знання для досягнення вибраної мети.

Ця низка термінів видається дуже важливою для інформатики та суспільства в цілому. Вона фактично вказує на етапи розвитку інформатики та суспільства: спочатку досліджується зовнішня сторона інформації - тобто дані (пригадайте структури даних мов програмування, бази даних тощо), далі акцент переноситься на інформацію (інформаційні технології, інформаційне суспільство), після чого мають прийти технології обробки знань та бази знань (суспільство, засноване на знаннях). І хоча зараз на етапі лише частково розвиненого суспільства важко про це говорити, колись має з'явитися «розумне (мудре) суспільство».

Повертаючись назад до поняття проблеми, бачимо, що найважливішою властивістю проблем є властивість інформаційності, тобто та обставина, що для розв'язання проблеми необхідно визначити (обчислити) деяку нову інформацію (на підставі наявної інформації). Цю властивість проблем назвемо принципом інформаційності проблем. Оскільки поняття програми є діалектичним запереченням поняття проблеми, то властивість інформаційності проблем у сфері програм постає як властивість отримання нових даних на підставі існуючих даних. Таким чином, програми застосовуються для одержання деяких нових даних (на підставі наявних даних), які у предметній області інтерпретуються як деяка нова інформація, що використовується для розв'язання проблеми.

У математиці об'єкти, що обчислюють нові дані на підставі вхідних даних, називають функціями, а сам процес застосування функцій до даних - аплікативністю.

Отже, основною (істотною) властивістю програм є властивість аплікативності (функціональності): програми застосовують до вхідних даних для одержання результатів. Внутрішня властивість аплікативності знаходить своє зовнішнє відображення у відношенні прагматичності програм, оскільки вона задає спосіб використання програм користувачем. Ця властивість дозволяє на гранично високому рівні абстракції трактувати програми як функції, які переводять вхідні дані у вихідні. Сформулюємо таке розуміння програм у вигляді наступного принципу.

Принцип аплікативності (функціональності). На гранично високому рівні абстракції програми можуть розглядатися як функції, які при застосуванні до вхідних даних можуть виробляти вихідні дані.

Отже, уточнення поняття програми починається з поняття даного і продовжується його запереченням - поняттям функції. Пов'язуються ці поняття аплікацією (застосуванням функції до даного, у результаті якого утворюється вихідне дане). Тут не потрібно, щоб результат застосування функції до даного був

завжди визначений або щоб результат такого застосування був однозначний. Інакше кажучи, будемо розглядати клас часткових багатозначних (недетермінованих) функцій. Такі функції задають зміст програм, який називається їх семантикою. Таке тлумачення семантики програм є найбільш абстрактним, бо згідно принципу розвитку ми починаємо з найбільш абстрактних формулювань.

Згідно з триадою розвитку тепер потрібно побудувати поняття, що є синтезом понять даного (теза) та функції (антитеза). Позначимо це нове поняття через X. Тоді можна записати наступне понятійне рівняння:

$$X = \text{синтез (теза: дане, антитеза: функція)}$$

Щоб з'ясувати властивості нового поняття, скористаємося законом заперечення. З триади розвитку випливає, що функція є діалектичним запереченням даного, а X – діалектичним запереченням функції, тобто X є заперечення заперечення даного. Це означає, що нове поняття X, з одного боку, зберігає функцію, а з іншого боку, за законом заперечення заперечення, підпадає під поняття даного, тим самим розвиваючи і це поняття. Говорючи більш просто, маємо, що X поводить ся як дане (з ним працюють як з даним), але X також має певний зв'язок із функцією (містить, задає функцію).

Щоб зрозуміти, що є X, пригадаймо, що зараз розвиваються перші, найабстрактніші аспекти поняття програми. Тому слід очікувати, що ці аспекти, зокрема і X, мають з'являтися на перших кроках роботи з програмами, зокрема і на екрані комп'ютера. Дійсно, якщо ми включимо комп'ютер, то відразу ж на його робочому столі побачимо певний набір іконок, ярликів. Але ж це і представники поняття X. Дійсно, кожна іконка є певним знаком, іменем програми, яку можна при потребі виконати. Таким чином, розв'язком наведеного понятійного рівняння є поняття імені функції. Відзначимо, що можна було б це поняття назвати знаком, символом, позначенням функції, але термін «ім'я» видається найбільш вдалим, тому що активно використовується у програмуванні та має давню традицію використання в філософії. Зокрема, перше речення (!) праці Аристотеля «Категорії» присвячене розгляду властивостей імен.

Зазвичай, під ім'ям розуміється такий об'єкт, з яким зіставляється деякий інший об'єкт, який називається денотатом (значенням) імені. Між іменем та значенням визначають різні відношення денотації. Якщо нас цікавлять відношення «значення – ім'я», то їх називають відношеннями іменування, номінації, для відношень ім'я – значення використовують терміни «розіменування», «деномінація», «денотація». З іменами ми поведимось як із даними. Імена складаються з певних символів, вони щось репрезентують (іменують, позначають).

На отриманому рівні абстракції можна сформулювати новий принцип.

Принцип номінативності програм. Програми можна представляти як імена, що позначають функції, які відображають вхідні дані в результати.

Поняття функції, імені і денотації, які є дуже абстрактними, мають широке коло

застосування. Так наприклад, у книгах з мов і систем програмування основна частина тексту присвячена функціям, їх діям і іменам, які дозволяють через відношення денотації працювати з функціями і розрізняти їх.

До цього моменту розвитку був встановлений зв'язок між даними і функціями за допомогою аплікації і між функціями та іменами функцій за допомогою відношення денотації. А як же пов'язані дані та імена функцій? Їх зв'язок задається похідним відношенням інтерпретації, що за ім'ям функції та даним визначають результат застосування функції, що позначається цим іменем, до обраного даного. Відзначимо, що якщо зазначену впорядковану пару (ім'я, дане) розглядати як структуроване дане, то операція (відношення) інтерпретації підпадає під поняття функції, що дозволяє будувати «універсальні» операції інтерпретації як «звичайні» функції.

Додатковий аспект операції інтерпретації стосується її обчислюваності. Подальший розвиток цих аспектів призводить до побудови інтерпретаторів мов програмування і, зокрема, комп'ютерів.

Таким чином, побудована триада понять, у якій в абстрактній формі виражені істотні аспекти програм. Ця триада складається з понять даного, функції й імені функції, пов'язаних певними відношеннями, основними з яких є аплікація, денотація та інтерпретація. Будемо називати її триадою основних програмних понять (рис. 2.8).

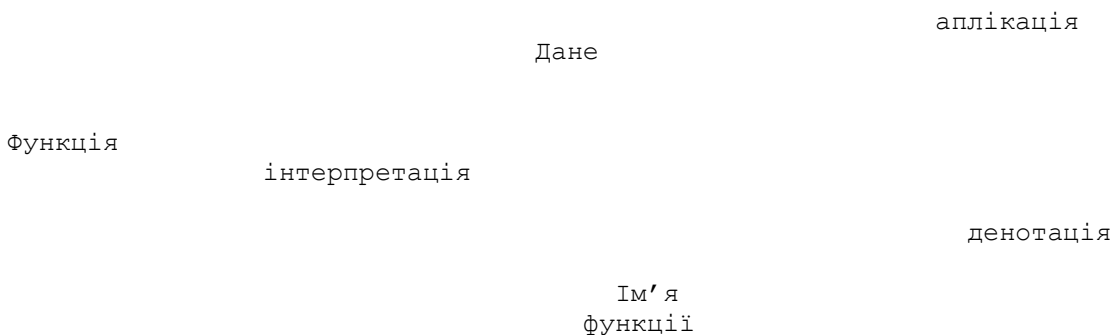


Рисунок 2.8. Триада основних програмних понять

У цій триаді всі поняття пов'язані одне з одним вказаними відношеннями, але і самі відношення пов'язані одне з одним. Наприклад, якщо нам задані відношення інтерпретації та аплікації, то можна визначити відношення денотації. Сформулюємо це більш точно. Нехай відношення інтерпретації $int(name, d, d')$ означає, що для імені функції $name$ та вхідного даного d вихідним даним буде d' , а відношення $appl(func, d, d')$ означає, що аплікація функції $func$ до вхідного даного d дає вихідне дане d' . Тоді імені функції n буде відповідати така функція f , що має місце рівність $int(n, d, d') = appl(f, d, d')$ для всіх даних. Власне кажучи, такий спосіб визначення денотації і використовується при операціональному підході до семантики програм.

Надалі при розвитку програмних понять будемо «повторювати» програмну триаду,

але на більш багатих рівнях.

2.4.2. Пентада основних програмних понять

Останнім моментом розвитку понять у триаді основних програмних понять було заперечення функції та перехід до імені функції, що зберігає функції за допомогою операції денотації. Для подальшого розвитку програмних понять необхідно зробити наступне заперечення, що повертає нас до поняття функції, але вже на новому рівні.

Відповідно до законів розвитку це означає, що ми переходимо до нового поняття, що не є іменем (але зберігає його) і яке підпадає під поняття функції, розвиваючи його. Позначаючи це нове поняття через Y , можемо записати наступне понятійне рівняння:

$$Y = \text{синтез (теза: функція, антитеза: ім'я функції)}$$

З триади розвитку випливає, що ім'я функції є діалектичним запереченням функції, а Y - діалектичним запереченням імені функції, тобто Y є заперечення заперечення функції. Говорячи більш просто, маємо, що Y поводить ся як функція (з цим поняттям працюють як із функціями), але також Y має певний зв'язок з іменами функцій та пов'язаним із ними функціями.

Аналогічно тому, як функції у програмній триаді є функціями над даними, цей новий об'єкт буде функцією над функціями, що мають імена (іменованими функціями). Такі об'єкти називаються композиціями функцій. Отже, з математичної точки зору композиції - це оператори (операції над функціями). Композиції і є новим поняттям Y . У композиціях операція іменування (денотації) повторює себе багато разів, тому що кожний новий аргумент композиції є новим набором іменованих функцій. Також і аплікація повторюється на новому рівні як застосування композицій до іменованих функцій. Отже, композиції розкривають структури функцій та з змістовної точки зору є засобами побудови функцій (програм). На новому рівні абстракції можна сформулювати наступний принцип [10].

Принцип композиційності. Програми (функції) будуються з більш простих програм за допомогою композицій.

У якості одного з наслідків цього принципу одержуємо, що повинні існувати деякі базові функції (програми), які використовуються для побудови більш складних функцій.

Для того щоб просуватися вперед, необхідно зробити нове заперечення і відповідно до закону заперечення заперечення буде здійснене повернення до поняття імені функції, але вже на новому рівні, який враховує наявність композицій. Нове поняття Z задається понятійним рівнянням

$$Z = \text{синтез (теза: ім'я функції, антитеза: композиція)}$$

Нові об'єкти із Z , які виникають, за своєю суттю є складними іменами, що описують функції. Такі об'єкти назовемо дескрипціями. Дескрипції фактично є текстами програм. Це дозволяє сформулювати наступний принцип.

Принцип дескриптивності. Програми можна представляти у вигляді дескрипцій (складних зображень), побудованих з використанням дескрипцій більш простих програм і композицій, та які описують функції, що відображають вхідні дані в результати.

Відзначимо етимологічну близькість терміна «дескрипції» терміну «програма». Обидва слова походять від дієслова «писати, малювати», тільки слово «програма» походить від грецького, а «дескрипція» – від латинського слова.

Дескрипції можуть розглядатися як імена, побудовані за допомогою більш простих імен. Правила побудови таких імен (дескрипцій) звичайно називаються граматикою. Граматика задає закони побудови правильних синтаксичних конструкцій. Стосовно предмета нашого дослідження граматика індуктивно задає правильні дескрипції (записи, зображення) на основі попередньо заданих імен базових функцій і імен (записів, зображень) композицій. Поняття дескрипції вже є досить багатим поняттям, оскільки допускається можливість виділення складових частин дескрипції та з'єднання таких частин у нову дескрипцію. Це фактично означає, що на класі дескрипцій повинні бути задані спеціальні операції, які дозволяють здійснювати таку роботу з дескрипціями. Відображення денотації для дескрипцій звичайно будуються індуктивно відповідно до структури дескрипції. Індуктивні визначення дескрипцій індукують індуктивні доведення різних властивостей програм.

У процесі розвитку поняття програми були введені п'ять програмних понять: дане

– функція – ім'я функції – композиція – дескрипція. Ці п'ять програмних понять описують на абстрактному рівні основні властивості програм. Будемо називати ці поняття з їх відношеннями пентадою основних програмних понять (рис. 2.9).

Ця пентада була утворена двома обертами розвитку. Перший оберт задається тріадою «користувач – проблема – програма». Другий оберт вводить поняття композиції та дескрипції, розгортаючи поняття функції та імені відповідно.

В пентаді чітко просліджуються три напрямки (три програмні аспекти):

- програмний аспект, що пов'язує дані, функції та композиції, який називають семантичним аспектом;
- програмний аспект, що пов'язує дані, імена та дескрипції, який називають синтаксичним аспектом;
- програмний аспект, що пов'язує імена та дескрипції з функціями та композиціями, який називають денотаційним аспектом.

Зауважимо, що для семантичного аспекту існують різні тлумачення: вузьке

тлумачення розглядає семантичний аспект як область значень (сміслів) програм, а широке тлумачення додатково включає в семантику зв'язок синтаксичного аспекту із смислами програм.

Рисунок 2.9. Пентада основних програмних понять

Введених понять достатньо для побудови багатьох моделей програм. Однак попередньо обговоримо питання про зв'язок цих понять із поняттям мови.

2.5. Сутнісні та семіотичні аспекти програм

Одним з основних програмних понять є поняття імені (знаку). Питаннями функціонування знаків займається також семіотика – наука про знаки та знакові системи. Тому не дивно, що здобутки семіотики застосовують і для визначення програмних аспектів.

Основні семіотичні аспекти (прагматика, семантика і синтаксис) були визначені та досліджені в роботах Ч. Пірса, Ч. Морріса, Р. Карнапа та інших учених:

- прагматика – це відношення між мовою і користувачем мови;
- семантика – це відношення між виразами мови та їх значеннями (денотатами), абстраговане від користувача;
- синтаксис – це відношення між мовними виразами, абстраговане від їх денотатів.

Ці три аспекти також застосовують до програм, визначаючи таким чином прагматичний, семантичний і синтаксичний аспекти програм. Будемо називати ці аспекти семіотичними аспектами програм.

Разом із тим побудовані пентади основних понять програмування та основних програмних понять дають дещо іншу класифікацію аспектів поняття програми.

Зокрема, перша пентада задає чотири основних зовнішніх аспекти:

- адекватнісний;
- прагматичний;
- обчислювальний;
- генетичний.

Ці аспекти розглядають програму як цілісність, не розкриваючи її структури.

Друга пентада дає три основних внутрішніх аспекти програм:

- семантичний;
- синтаксичний;
- денотаційний.

Ці дві пентади розкривають зовнішні та внутрішні аспекти програм. Необхідно

також продемонструвати зв'язок зовнішніх та внутрішніх аспектів. Так, функція до даного (яка є внутрішнім аспектом) має зовнішнє проявлення у застосуванні програм до даних (обчислюваності програм). Внутрішнє поняття композиції розкривається у зовнішньому відношенні експлікативності як декомпозиція програми. Інші введені відношення також пов'язані між собою.

Вказані аспекти розкривають зовнішні і внутрішні властивості програм та їх зв'язок. Це відповідає категорії сутності, тому будемо називати наведені сім аспектів сутнісними аспектами програм.

Таким чином, тут уведено два підходи до розкриття основних програмних аспектів: семіотичний та сутнісний. Доцільно зробити порівняльний аналіз запропонованих підходів.

Таблиця 2.1

№ підхід	Характеристика підходу	Семіотичний підхід	Сутнісний
1	Рівень орієнтація на загальності розглядаються на абстракції.	Орієнтація на довільні системи, що використовують знаки.	Більш спеціальна програми, які високому рівні
2	Основні поняття десять, точно виразити підходу не надаючи одному з них.	Основним поняттям є знак, додатковими – користувач та значення. Тому головний акцент – на знаках, невелика кількість понять не дає можливості визначати достатню кількість аспектів.	Основних дає змогу більш різні аспекти, перевагу лише
3	Класифікація класифікуються на три аспекти	Класифікація аспектів відсутня. Тому чіткого розподілу на зовнішні та внутрішні аспекти немає. До зовнішніх можна віднести лише прагматичний аспект, частково – семантичний аспект, до внутрішніх – синтаксичний та семантичний (частково).	Аспекти чотири зовнішніх та внутрішніх.
4	Зовнішні зовнішні аспекти чіткими дозволяє вивчення	До зовнішніх можна віднести лише прагматичний аспект. Як наслідок, це веде до перенавантаження прагматичного аспекту, який стає занадто розпливчастим та нечітким.	Виокремлено чотири аспекти з достатньо визначеннями. Це розгорнути детальне цих аспектів.
5	Внутрішні віднесено аспекти синтаксичний та аспекти. аспекти в відповідають такі сутнісному підході: теорія денотаційний, який	До внутрішніх можна віднести семантичний (частково) та синтаксичний аспекти. Але семантичний аспект визначає відношення знака до його значення, тобто має синтактико-семантичний напрямок, що ставить	До внутрішніх семантичний, денотаційний Семантичному семіотиці аспекти в семантичний, як значень, та

<p>(тобто синтаксис. Це розглядати синтаксичні підходи визначень саме знаходиться у стані.</p>	<p>синтаксис у привілейоване становище.</p>	<p>пов'язує значення семантику) та дає можливість семантико-до розгортання програм, у яких семантика привілейованому</p>
--	---	--

Як видно з наведеної таблиці, теорія семіотичних аспектів трактує програму як текст (зображення). На наш погляд, таке трактування є занадто бідним для програмування, бо залишає поза увагою багато важливих аспектів програм та робить центральним тлумачення програми як знака. На противагу семіотичному підходу сутнісна теорія програмних аспектів тлумачить програму як єдність її різних сторін, як єдність форми та змісту. Тому семантичний, синтаксичний та денотаційний аспекти стають внутрішніми аспектами програм, а прагматика, адекватність, обчислюваність та генетичність - зовнішніми, що дозволяє будувати більш адекватні теорії структур програм та програмування.

Порівнюючи семантику та синтаксис із категоріями змісту та форми, можна зробити висновок про те, що семантика є провідним аспектом, а інші аспекти можуть розглядатися як похідні. Це дозволяє сформулювати наступний принцип.

Принцип підпорядкованості. Семантика є провідним внутрішнім аспектом програм, а синтаксичний та денотаційний аспекти підпорядковані семантиці [10].

Всі аспекти варто спочатку вивчати незалежно один від одного, а потім - у їх єдності.

Внаслідок принципу підпорядкованості впливає, що семантичний аспект є важливіший за синтаксичний. Звідси впливає, що будемо використовувати семантико-синтаксичний підхід до дослідження та формалізації програмних понять. Сформулюємо це у вигляді наступного принципу.

Принцип семантико-синтаксичної орієнтації. Основним підходом до дослідження та формалізації програмних понять буде семантико-синтаксичний підхід, а синтаксико-семантичний підхід будемо розглядати як допоміжний.

Аналізуючи семантичний та синтаксичний аспекти, можна стверджувати, що для семантичного аспекту найважливішим є поняття композиції, бо саме воно визначає властивості програм. Крім того, відзначаємо також важливість відношень іменування (номінативність). Це дозволяє називати запропонований підхід композиційно-номінативним.

Сформульовані принципи та пентада програмних понять утворюють базис композиційно-номінативного підходу до уточнення поняття програми. Використання терміна «композиційний» у назві запропонованого підходу можна пояснити кількома причинами. Серед трьох термінів: «дані», «функція» і «композиція» – користувачі зацікавлені насамперед у даних і функціях. Однак, щоб зрозуміти властивості даних і функцій, необхідно вивчити властивості композицій, які визначають структури (а отже, властивості) функцій і даних. Тому основна увага буде приділена вивченню композицій та їх властивостей, у той час як властивості функцій і даних будуть похідними від властивостей композицій. Інакше кажучи, композиції задають логічну складову програмування, а функції та дані – предметну. Що ж стосується терміна «номінативний», то його використання в назві підходу підкреслює фундаментальну роль відношень іменування (денотації) у побудові всіх програмних понять.

Таким чином, серед різних парадигм програмування (процедурна, функціональна, алгебраїчні, логічна тощо), які відображають різні аспекти програм, будемо вивчати парадигми композиційності і номінативності в межах композиційно-номінативного підходу. Цей підхід може бути охарактеризований як семантико-синтаксичний підхід, орієнтований на систематичне вивчення номінативних відношень у побудові даних, функцій, композицій і дескрипцій.

2.6. Програми і мови

Поняття програми тісно пов'язане з поняттям мови. Дескрипції можуть трактуватися як речення мови, функції – як значення речень. Тому способи подання програм можуть розглядатися як певні мови, які називають мовами програмування. Для дослідження таких штучних мов можна використовувати лінгвістику, яка була розвинена для природних мов. Разом із тим є певні відмінності між природними мовами та мовами програмування, які сформулюємо у наступній таблиці. Зауважимо, що, використовуючи термін «мова програмування», ми фактично будемо більше говорити про мови програм, ніж про процеси програмування.

Таблиця 2.2

№ програмування мови	Характеристика	Природні мови	Мови
1	Походження програмування є	Природні мови з'явилися у результаті історичного розвитку певної групи людей.	Мови штучними мовами.
2	Основні аспекти програмування, достатньо точні	Природні мови мають багато основних аспектів, таких як комунікативний, когнітивний,	Основних але вони мають визначення.

		референціальний, сігніфікативний, експресивний, лікувальний і т.д.	
3	Точність програмування, як визначення визначені. мови	Точних визначень немає, є різні моделі. Мови багатозначні.	Мови правило, точно Мова однозначна.
4	Відкритість	Мова постійно розширюється.	Мова фіксована.

Як бачимо, мови програмування значно простіші та бідніші за природні мови. Але це дає можливість більш точно описати мови програмування. Однозначність мов програмування важлива при створенні відповідних систем програмування та при дослідженні програм.

2.7. Пентада програмних понять процесного типу

Введені програмні поняття є досить абстрактними. Згідно з принципом розвитку від абстрактного до конкретного, треба розвивати основні поняття до більш конкретних та багатих.

Наступним рівнем конкретизації може бути розгляд більш конкретних типів проблем не лише функціонального типу, як це було зроблено раніше, але і проблем процесного типу, коли їх суть полягає не стільки в інформаційності, скільки в необхідності виконання низки дій. Такі проблеми будемо називати проблемами процесного типу.

Уточнення програмних понять для проблем процесного типу веде до пентади, аналогічної раніше побудованій, у якій замість поняття функції береться поняття процесу.

Отримуємо наступну пентаду: дія - процес - ім'я процесу - композиція процесів - дескрипція процесів, яка подана на рис. 2.10.

Поняття процесу є важливим для моделювання та дослідження більш багатих програмних систем, а саме розподілених, паралельних, реактивних тощо. В межах першої частини посібника такі системи розглядати не будемо.

Рисунок 2.10. Пентада програмних понять процесного типу

Висновки
У цьому розділі було розвинено основні поняття програмування на основі загальногносеологічних принципів, головним з яких є принцип розвитку від абстрактного до конкретного. Основні результати можна сформулювати таким чином.

1. Було розглянуто словникові визначення поняття програми та

продемонстровано, що ці визначення нечіткі, спрощені та ведуть до хибного кола визначень.

2. Були розглянуто основні гносеологічні принципи визначення понять. За основу був узятий підхід, що базується на діалектичній логіці, але береться у спрощеному вигляді для розгляду лише проблем програмування.

3. Продемонстровано важливість трьох взаємопов'язаних рівнів розгляду понять, а саме методологічного (філософського), професійного (конкретно-наукового) та формального (математичного).

4. Було розвинено основні поняття програмування в їх взаємозв'язках: спочатку триада «користувач – проблема – програма», яка потім у пентаді доповнюється поняттями процесу виконання та процесу програмування. Введені відношення задають зовнішні сторони понять програми та програмування.

5. Було розвинено основні програмні поняття в їх взаємозв'язках: спочатку триада «дане – функція – ім'я» функції, яка потім в пентаді доповнюється поняттями композиції та дескрипції.

6. Основними внутрішніми аспектами програм є семантичний, синтаксичний та денотаційний аспекти, які тісно пов'язані із зовнішніми аспектами.

7. Було зроблено порівняльний аналіз семіотичних та сутнісних аспектів програм.

Семіотичні аспекти (прагматика, семантика, синтаксис) ставлять лише поняття

знака у центр уваги, применшуючи важливість інших понять. Теорія сутнісних

аспектів програм виділяє зовнішні (адекватність, прагматичність, обчислюваність,

генетичність), внутрішні аспекти (семантика, синтаксис, денотація) та їх зв'язки,

дозволяючи більш адекватно розкрити поняття програми.

8. Зроблено порівняльний аналіз природних мов та мов програмування.

Підкреслено вимоги однозначності та точності для мов програмування.

9. У контексті подальшої конкретизації основних понять програмування

побудовано пентаду понять для проблем процесного типу.

Додаткову інформацію про основні поняття програмування можна найти у [2-4,8,9].

Контрольні питання

1. Проаналізуйте визначення поняття програми та програмування, наведені у різних словниках.
2. Розкрийте зміст принципу гносеологічності.
3. Наведіть приклади категорій.
4. Розкрийте зміст принципу програмологічної проекції.
5. Розкрийте зміст категорій «абстрактне» та «конкретне».
6. Як формулюється принцип розвитку?
7. Як розуміється відносний характер принципу розвитку та інших гносеологічних принципів?
8. Які поняття та відношення утворюють початкову триаду понять програмування?
9. Які поняття та відношення утворюють триаду прагматичності програм?
10. Які поняття та відношення утворюють триаду основних понять програмування?

11. Які поняття та відношення утворюють пентаду основних понять програмування?
12. Наведіть визначення програмування різних рівнів абстракції.
13. Які поняття та відношення утворюють тріаду основних програмних понять?
14. Розкрийте зміст принципу інформативності проблем.
15. Як формулюється принцип аплікативності?
16. Як формулюється принцип номінативності?
17. Як формулюється принцип композиційності?
18. Як формулюється принцип дескриптивності?
19. Які поняття та відношення утворюють пентаду основних програмних понять?
20. Сформулюйте семіотичні аспекти програм.
21. Сформулюйте сутнісні аспекти програм.
22. Як формулюється принцип підпорядкованості?
23. Як формулюється принцип семантико-синтаксичної орієнтації?
24. Які властивості формулюються для природних мов та мов програмування?
25. Які поняття та відношення утворюють пентаду основних програмних понять процесного типу?

3. Формалізація програмних понять

Перейдемо до побудови математичних уточнень введених програмних понять – до їх формалізації.

Згідно з принципом розвитку від абстрактного до конкретного, можна побудувати різні формалізації поняття програми. Найпростіша формалізація відповідає тріаді основних програмних понять: дане – функція – ім'я функції. Задаючи деякий клас даних D , програму можна уточнити як пару (fn, f) , де fn – ім'я програми, а $f : D \rightarrow D$ – функція, що задає семантику програми. Клас програм тоді задається як певний клас таких пар і називається інтерпретованим класом програм.

Пентада програмних понять дозволяє побудувати більш конкретні та багаті формалізації для програм, що будуть враховувати їх композиційну структуру та структуру дескрипцій. У цьому випадку можна говорити про те, що класи програм задаються мовою програмування (програмною системою), яка може бути подана як трійка систем, що задають семантику, синтаксис та відношення денотації.

Які ж формалізми можна використовувати для подання програмних систем?

3.1. Теоретико-функціональна формалізація

Традиційно для формалізації поняття програми використовують теоретико-множинний підхід, часто в дуже специфічній формі. Наприклад, такі методи формальної розробки програм, як В [11] та Z [16], безпосередньо спираються на аксіоматичну теорію множин Цермело-Френкеля. Тому часто під формальною семантикою розуміють теоретико-множинну семантику. Водночас, незважаючи на наявний багатий позитивний досвід теоретико-множинних формалізацій, починаються спроби проводити формалізацію програмних понять не тільки на основі поняття множини, а на її основі поняття функції (відображення). Стосовно поняття програми, теоретико-функціональний підхід підтримується принципом

аплікативності (функціональності) програм. У межах композиційно-номінативного підходу також будемо орієнтуватися на теоретико-функціональну формалізацію. Відразу ж відзначимо, що поняття функції і множини нерозривно пов'язані одне з одним, тому коли мова йде про використання теоретико-функціонального підходу то основна увага буде приділятися поняттю функції, а не поняттю множини. У зв'язку з цим сформулюємо наступний принцип.

Принцип теоретико-функціональної формалізації. Формалізація поняття програми здійснюється в межах теоретико-функціонального підходу, в основі якого лежить поняття функції (відображення), яке має розглядатися на різних рівнях абстракції.

Сформульований принцип має дві сторони. Перша полягає в тому, що більш адекватним для формалізації поняття програми є теоретико-функціональний підхід, а не теоретико-множинний, який розглядає лише екстенціональні аспекти функцій. Друга сторона принципу полягає в тому, що треба збагачувати існуючі визначення функцій, вводячи інтенціональні аспекти та властивості функцій і даних різних рівнів абстракції. Отже, мова йде про те, щоб зблизити поняття програми та функції, збагативши тлумачення функцій тими інтенціональними аспектами, які є у програм.

З огляду на програмістську орієнтацію підходу, функції будуть у першу чергу трактуватися аплікативно (операціонально), як деякі абстрактні пристрої (машини, автомати), які при застосуванні до вхідних даних, можуть виробляти результати.

Таким чином, у процесі формалізації будемо вводити об'єкти різного рівня абстракції, які належать деякому універсуму об'єктів. На граничному рівні абстракції об'єкти розглядаються як чорні скриньки. На наступному рівні абстракції серед цих об'єктів виділяються спеціальні об'єкти, які називаються функціями (відображеннями). Ці об'єкти трактуються як деякі машини, які мають вхідний порт (канал), на який можуть надходити дані, та вихідний порт (канал), з якого зчитуються результати. Передбачається, що результати можуть бути отримані за скінченний час. Якщо за скінченний час результату немає, то результат застосування машини (функції) до вхідного даного покладається не визначеним. На верхньому рівні абстракції внутрішній устрій машини (функції) невідомий (машина є чорною скринькою з двома портами), на більш низьких рівнях абстракції структура може бути частково розкрита (сірий ящик). Для деяких функцій або об'єктів їх внутрішній устрій цілком розкритий (біла скринька). Як наслідок будемо використовувати різні типи машин для опису класів відображень.

3.2. Класи функцій

Введемо визначення та позначення для різних класів функцій. Будемо розглядати найбільш загальний клас часткових багатозначних функцій. Визначення стосуються екстенціональних аспектів функцій, тобто тих аспектів, які задаються лише через аргументи та значення функцій.

Нехай D і R - деякі класи об'єктів. Клас об'єктів, розглянутих як часткові

m

багатозначні функції з D у R , позначимо $D \multimap R$. Для $f \in (D \multimap R)$ позначимо $\text{arg}(f)$

клас визначеності f , тобто клас об'єктів, на яких f може бути визначена, $\text{und}(f)$ – клас

невизначеності f , тобто клас об'єктів, на яких f може бути не визначена, $\text{gr}(f) \subseteq D \times R$ –

графік f , тобто множина пар (d, r) таких, що значення f на d може бути визначеним

із значенням r . Відзначимо, що перетин $\text{arg}(f) \cap \text{und}(f)$ може бути не порожнім,

оскільки можуть бути об'єкти, на яких f може бути як визначена, так і не визначена.

Якщо ж $\text{arg}(f) \cap \text{und}(f) = \emptyset$, то f будемо називати реляційною функцією, оскільки в

такому разі вона однозначно визначається своїм графіком, тобто бінарним відношенням (реляцією) на $D(R)$.

Для подання функцій будемо часто використовувати індексний конструктор функцій виду $[d \ a \ r \mid i \in I]$ або $[d : D \ a \ r : R \mid i \in I]$, де $d \ i \ r$ можна трактувати як

$i \quad i \quad i \quad i \quad i$

індексні часткові однозначні відображення з деякої множини індексів I у класи $D \ i \ R$

m

відповідно. Такий конструктор задає функцію $f \in D \multimap R$ наступним чином. На

$d \in D$ значення f може дорівнювати значенню r для такого $i \in I$, для якого d визначено

i

i дорівнює d . Якщо ж таке r_i не визначено або ж необхідного i не існує, то значення f

на d може бути не визначене.

Клас часткових однозначних функцій із D у R позначимо $D \rightarrow R$, клас тотальних

(усюди визначених) багатозначних функцій – $D \multimap R$, тотальних однозначних функцій – $D \rightarrow R$.

Введені позначення класів функцій дозволяють

перейти до формалізації програмних понять. Формалізми, які виникають будемо називати програмними системами.

3.3. Програмні системи

Програмні системи – це специфічна форма фіксації рівня абстракції програмних понять. Вона дозволяє визначити сукупність понять, яка характеризує рівень, та їх

властивості, що задаються сукупністю суджень.

Програмні системи будуть визначатися відповідно до схеми розвитку програмних понять, що задається

програмною пентадою. Кожне поняття представляється відповідною системою, яка фіксує зв'язок розглянутого поняття з іншими поняттями. Залежно від рівня

абстракції такі системи можуть бути простими або дуже складними. Фактично

повинна бути побудована мережа взаємопов'язаних систем.

Як ілюстрацію дамо визначення програмних систем, які знаходяться на гранично високому рівні абстракції.

Починаємо з системи даних. Оскільки ніяких властивостей даних не задано, то система даних D_t просто визначає деякий клас D , тобто $D_t = \langle D \rangle$. Наступне поняття в

пентаді - функція. Система функцій F_n - це пари $\langle D, F \rangle$, де $F \subseteq D \rightarrow D$. Імена функцій задаються як деяка множина Z , тому система імен N_m - це одноелементний кортеж $\langle Z \rangle$. Відношення денотації (номінації) будемо трактувати як відображення

$den: Z \rightarrow F$. Тому система денотації D_n є кортеж $\langle den \rangle$. Остання трійка систем

визначає в абстрактному виді семантику програм (система функцій), їх синтаксис

(система імен) та зв'язок синтаксису із семантикою (система денотації). Тому

доцільно ввести нову - функціонально-номінативну - систему, яка задає ці системи в

їх єдності. Функціонально-номінативна система FNS - це кортеж $\langle D, F_n, N_m, D_n \rangle$.

Подальший розвиток буде здійснюватися конкретизацією введених систем.

Відзначимо, що функціонально-номінативна система може тлумачитись як

алгебра даних: D - носій цієї алгебри, а операціями алгебри є функції, які задаються

іменами з Z за допомогою відображення den . Далі це поняття буде розвинене до алгебри функцій.

При побудові таких систем виникає питання про форми їх подання. Наприклад, функціонально-номінативну систему FNS можна було б представити не у вигляді «ієрархічного» кортежу $\langle D, F_n, N_m, D_n \rangle$, а у вигляді «плоского» кортежу $\langle D, F, Z, den \rangle$.

Кожна з форм уявлення може бути зручної в тих або інших випадках. Тому, щоб не обмежувати себе однією формою уявлення, приймемо

наступну угоду про

множинність форм подання програмних систем: залежно від цілей дослідження

програмні системи можуть представлятися в різних формах.

Наступне поняття в пентаді - композиція, яке уточнимо в межах композиційної

системи $C_m(Z) = \langle D, F, C \rangle$, де $C \subseteq (Z \rightarrow F) \rightarrow F$. Тут Z виступає як множина імен аргументів композицій.

Нарешті, формалізуємо поняття дескрипції. Насамперед зазначимо, що наявність композицій дозволяє ієрархічну побудову програм із деяких базових функцій

багатократним застосуванням композицій. Тому дескрипції визначаються

індуктивно. Інша обставина - дескрипції також будемо трактувати як відображення відповідно до теоретико-функціонального підходу. Тому для побудови

дескрипцій необхідні множини імен базових функцій FN , композицій

CN і аргументів

композицій Z . Множина інфінітарних стандартних дескрипцій $StDes$ задається індуктивно:

1) $StDes(0) = FN;$

$$(i+1) \qquad \qquad \qquad m \qquad \qquad \qquad (i)$$

2) $StDes = \{ [cnads] \mid cn \in CN, ds \in Z \mapsto StDes \}, i \in \omega.$
Тоді $StDes = \bigcup_{i \in \omega} UStDes(i).$

Наведене визначення використовує довільні відображення імен аргументів у множину раніше побудованих дескрипцій. Це дозволяє задавати дуже загальні, в тому числі й інфінітарні (нескінченні) дескрипції. У програмуванні звичайно обмежуються використанням фінітарних дескрипцій. Для виділення спеціальних класів стандартних дескрипцій уведемо відповідну програмну систему $Ds(Z) = \langle FN, CN, Ds \rangle$, де $Ds \subseteq StDes$, яку назвемо стандартною дескриптивною системою.

Відношення денотації конкретизується як відображення $den: Ds \mapsto F$. Внаслідок індуктивної побудови дескрипцій досить задати відношення денотації для

імен базових функцій та імен композицій: $den_fn : FN \mapsto F$ та $den_cn : CN \mapsto C$.

Тоді значення den на $ds \in Ds$ задається індуктивно:

- 1) якщо $ds = fn$, $fn \in FN$, то $den(ds) = den_fn(fn)$,
 - 2) якщо $ds = [cnads]$, де $ds = [z ads \mid z \in Z, ds \in Ds, i \in I]$,
- $$i \qquad i \qquad i \qquad i$$
- то $den(ds) = (den_cn(cn))([z aden(ds) \mid i \in I])$.

У цьому випадку систему денотації можна задати як пару $BDn = \langle den_cn, den_fn \rangle$. Тому функціонально-номінативна система конкретизується як композиційно-

дескриптивна система $CDS = \langle Cm(Z), Ds(Z), BDn \rangle$. Такі системи можуть тлумачитися як алгебри функцій, операціями яких є композиції.

Зазначимо, що можна розглядати нестандартні дескриптивні системи, але для того щоб визначити денотати таких дескрипцій будемо вважати, що є відображення нестандартних дескрипцій у стандартні. Таке відображення будемо називати відображенням аналізу, а дескриптивні системи із заданим відображенням аналізу будемо називати аналізованими дескриптивними системами.

Приведені уточнення програмних понять можна розглядати як перший, найбільш абстрактний оберт спіралі розвитку поняття програми. Подальші конкретизації фактично починають другий оберт спіралі.

3.4. Рівні конкретизації програмних систем

Розгляд рівнів почнемо з конкретизації поняття даного.

Рівні конкретизації даних. Як відзначалося раніше, треба будувати програмні поняття різних рівнів абстракції. Це стосується і даних. У цьому розділі розглянемо конкретизації даних, які фактично індуковані розвитком поняття даного в програмній пентаді (дане - ім'я - дескрипція). Інші конкретизації розглянемо пізніше. Програмна пентада фактично вказує на три рівні розгляду даних. Перший рівень - абстрактний, коли ніякі властивості даних не виділяються і дані розглядаються як елементи деякої абстрактної множини. Другий рівень визначається тим, що виділяються деякі специфічні дані. Звичайно такі дані застосовуються як логічні константи, тому назвемо другий рівень булевим рівнем. Нарешті, на третьому рівні є можливість виділяти компоненти даних. З огляду на те, що таке виділення компонент звичайно робиться за допомогою відношень іменування (номінації), назвемо третій рівень номінативним рівнем. Можна також охарактеризувати дані на абстрактному рівні як чорні (непрозорі) скриньки, константи булевого рівня - як білі (прозорі) скриньки, а дані номінативного рівня - як сірі (частково прозорі) скриньки.

Три виділені рівні приводять до трьох класів систем даних. Абстрактна система даних (визначена раніше) - $ADS = \langle D \rangle$. Система даних булевого рівня в загальному випадку має вигляд $BDS = \langle D, c_1, c_2, \dots \rangle$, де c_1, c_2, \dots - деякі виділені елементи (константи) з D . Можна виділити три типи систем: перед-булеву систему $\langle D, true \rangle$, булеву систему $\langle D, true, false \rangle$ та пост-булеву систему $\langle D, c, \dots, c_k \rangle$.

1

На номінативному рівні дані розглядаються як номінативні дані. Клас ND інфінітарних номінативних даних будується індуктивно на основі деяких множин

імен V і значень W :

$$1) ND(0) = W,$$

$$(i+1) \quad m \quad (i)$$

$$2) ND = \bigvee_{i \in \omega} ND(i), \quad i \in \omega.$$

$$\text{Тоді } ND = \bigcup_{i \in \omega} ND(i).$$

$$i \in \omega$$

Таким чином, номінативні дані фактично є ієрархічно побудованими відображеннями, областю визначення яких є множина імен, а областю значень - номінативні дані.

Система даних номінативного рівня може бути задана як кортеж $NDS = \langle V, W, D \rangle$, де $D \subseteq ND$.

Зазначимо, що вищенаведене індуктивне визначення номінативних даних може

м

бути подане у вигляді наступного рекурсивного визначення: $ND = W \cup (\bigvee_{i \in \omega} ND(i))$.

Ці визначення допускають інфінітарні (нескінченні) номінативні дані. У програмуванні клас номінативних даних обмежується класом ієрархічно побудованих скінченних реляційних відображень. Тому виділимо наступні основні

класи номінативних даних:

- однозначні номінативні дані (скінченні однозначні відображення), які називаються іменними даними;

- багатозначні номінативні дані (скінченні багатозначні реляційні відображення), які називаються просто номінативними даними.

Іменні дані дозволяють формалізувати структури даних, які використовуються у програмуванні та характеризуються однозначністю іменування своїх компонент. Це такі структури даних, як записи, масиви, файли, реляції тощо.

Номінативні дані дозволяють задавати структури даних, яким властива неоднозначність іменування

своїх компонент, такі як множини, перед множини, мультимножини тощо.

Наприклад, множину вигляду $\{s_1, s_2, \dots, s_n\}$ можна представити номінативним даним

1 2 n

$[1\ as_1, 1\ as_2, \dots, 1\ as_n]$, де 1 обрано як стандартне ім'я елементів множини. Для

1 2 n

роботи з такими даними повинні бути введені спеціальні функції, які відображають рівень абстракції даних. Важливо відзначити, що всі такі функції є конкретизаціями

абстрактних функцій опрацювання номінативних даних, що істотно зменшує

загальне число функцій та спрощує їх дослідження. Зазначимо також, що розгляд

даних як спеціальних функцій має давні традиції у програмуванні.

Рівні конкретизації функцій. Кожен із рівнів конкретизації даних індукує

відповідний рівень розгляду функцій. Абстрактному рівню відповідає абстрактний

(нетипізований) клас функцій з D у D . Булевому рівню відповідають такі класи, як

часткові характеристичні функції - часткові відображення на D , що приймають

лише значення `true` (перед-булевий рівень), предикати - часткові відображення з D у

`Bool` (булевий рівень), і часткові скінченнозначні відображення з D у $\{c_1, \dots, c_k\}$ (пост-

1 k

булевий рівень). Природно, що найбільш багаті класи функцій можна виділити на

номінативному рівні. Фактично такі класи визначають класи функцій над різними структурами даних. У цьому випадку системи функцій конкретизуються як

типізовані системи. Серед функцій номінативного рівня основними будуть різні

типізовані функції іменування, розіменування, накладення, видалення компонент

тощо.

Рівні конкретизації композицій. Як і в попередньому випадку, доцільно виділити три рівні композицій. Обмежимося тут розглядом композицій як однозначних відображень. У цьому випадку на абстрактному рівні можна виділити лише одну нетривіальну композицію – послідовне виконання • (яку звичайно називають функціональною композицією). Однак цього вже досить, щоб почати розвиток теорії категорій, для якої множення функцій є основною досліджуваною операцією. На булевому рівні можна додатково визначити такі композиції, як умовні оператори та цикли різних видів. Тому до булевого рівня можна віднести системи структурного програмування та системи алгоритмічних алгебр. Найбільш багатий композиціями номінативний рівень. Тут визначають наступні основні засоби конструювання програм: різні суперпозиції, розіменування, присвоювання тощо. Зазначимо, що введене поняття композиційної системи дозволяє дати опис повних класів композицій на різних рівнях абстракції, тобто описати всі композиції відповідного рівня.

Рівні конкретизації дескрипцій. Різноманіття типів даних, функцій та композицій індукує різні типізовані дескриптивні системи. Поряд із стандартними дескрипціями, визначеними раніше, використовуються спеціальні класи дескрипцій, зокрема, якщо композиції є скінченно-арними відображеннями, то імена їх аргументів є натуральними числами та їх впорядкованість можна використовувати для подання дескрипцій як послідовностей символів (слів у деякому алфавіті). Звичайно для визначення таких лінійно впорядкованих дескрипцій використовують породжуючі граматики. Ці граматики будуть детальніше розглянуті у наступному розділі.

Інтеграція композиційних і дескриптивних систем номінативного рівня за допомогою відображень денотації дозволяє дати цілісний опис різноманітних мов програмування і баз даних. Такі композиційно-дескриптивні системи номінативного рівня будемо називати композиційно-номінативними системами. Ці системи дозволяють також адекватно описувати семантику різних логічних мов і мов специфікацій прикладних областей.

Висновки

У розділі розглянуто підходи до формалізації програмних понять. Сформульовано принцип про важливість теоретико-функціональної формалізації поняття програми, при якій програми уточнюються як функції, що розглядаються на різних рівнях абстракції. Введено різні класи функцій. Розглянуто приклад формалізації програми. Введено поняття програмної системи та розглянуто системи різного рівня абстракції. Визначені програмні системи абстрактного, булевого і номінативного рівнів. Програмні системи номінативного рівня і різні їх конкретизації, які називаються композиційно-номінативними системами, є

основним формалізмом опису програм. Будучи порівняно простими в побудові, ці системи, проте, мають велику потужність і дозволяють адекватно уточнити і досліджувати основні властивості програм мов програмування.

Додаткову інформацію про програмні системи можна знайти у [2, 4, 8, 15].

Контрольні питання

1. Розкрийте зміст принципу теоретико-функціональної формалізації поняття програми.
2. Визначте різні класи функцій.
3. Визначте програмні системи різного рівня абстракції.
4. Дайте визначення класу номінативних даних.

4. Синтактика: формальні мови та граматики

4.1. Розвиток понять формальної мови та породжуючої граматики

Мови програмування є штучними мовами, спеціально створеними для запису програм. На відміну від природних мов, які є багатоаспектними, неоднозначними, відкритими, відносно швидко змінюються, штучні мови значно бідніші, від них вимагають фіксованості смислу та однозначності.

Синтаксичний аспект є одним з основних аспектів програм. За визначенням, він пов'язаний з формами подання програм в абстракції від інших їх аспектів.

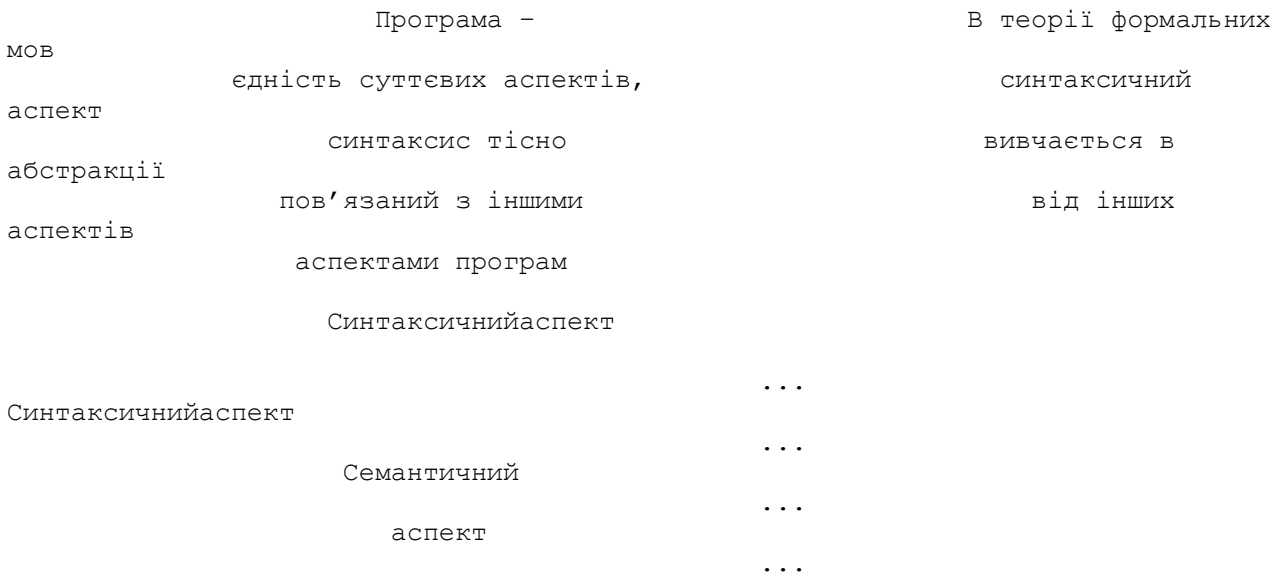


Рисунок 4.1. Абстрагування синтаксичного аспекту від інших аспектів програм

Момент абстракції синтаксичного аспекту від інших аспектів програм є дуже важливим. Це означає, що на першому етапі розвитку синтактики - науки про синтаксичний аспект - визначається головним чином лише відношенням знаків між собою. (див. рис. 4.1). Разом із тим не слід абстрагувати цей момент абстрагування від інших аспектів, бо на другому етапі синтаксис програм має бути узгоджений з іншими їх аспектами.

Наведені твердження сформулюємо як принцип абстрагування та єдності синтаксичного аспекту з іншими аспектами програм.

Уточнення синтаксичного аспекту штучних мов здійснюється на основі поняття формальної мови. Як уже говорилося, це абстрактне поняття, що відображає перші (синтаксичні) характеристики мови. Тому термін «формальний» тлумачиться не тільки як «точно заданий», але і як «абстрактний».

Хоча синтаксичний аспект пов'язаний з багатьма аспектами програм, найтісніший зв'язок є з семантичним аспектом. Зупинимось на цьому зв'язку детальніше.

Синтаксичний та семантичний аспекти є подальшою конкретизацією (проекцією) категорій форми та змісту. Тому, вивчаючи синтаксис (теорію формальних мов), ми все ж не повинні забувати, що він є похідним від семантичного аспекту, і саме ця обставина і визначає методи експлікації (уточнення) та напрями дослідження синтаксису. Щоб обґрунтувати вторинність синтаксичного аспекту, треба розглянути зовнішні аспекти програм, і в першу чергу аспект адекватності програми проблемі. Зрозуміло, що в проблемі нас цікавить не стільки її форма, як її зміст. Це фактично означає, що семантика буде провідним аспектом, а синтаксис – похідним від семантики.

Щоб зафіксувати ці положення у явній формі, сформулюємо їх у вигляді наступного принципу синтактики.

Принцип відокремлення, підпорядкування та єдності синтаксичного та семантичного аспектів: синтаксичний аспект програм є похідним від семантичного, він спочатку вивчається в абстракції від семантичного аспекту, а потім – у єдності з ним.

Наведений принцип фактично вказує лише на відношення синтаксичного аспекту до семантичного, але не дає ніякої характеристики синтаксичному аспекту.

Подальший крок конкретизації полягає в тому, що будемо уточнювати синтаксичний аспект за допомогою поняття формальної мови. Елементи, які задаються формальною мовою, будемо називати реченнями. Відзначимо, що тут можна було б вжити замість терміна «речення» терміни «слово», «текст», або інші терміни, що характеризують основні структурні елементи мови; цей вибір залежить від цільової направленості дослідження.

Проаналізуємо тепер, на якому рівні абстракції розглядаються речення. Згідно принципу розвитку від абстрактного до конкретного будемо спочатку речення тлумачити абстрактно як цілісності. Але цілісності якого типу – «білої» чи «чорної» скриньки?

Неважко зрозуміти, що елементи формальних мов – речення – слід розглядати як «білі скриньки», тобто їх можна розпізнавати та відрізняти одне від одного.

Обґрунтування вибору такого тлумачення надає принцип відокремлення, підпорядкованості та єдності, який стверджує вторинність синтаксису від семантики, зокрема і те, що синтаксис речення повинен допомогти у встановленні його семантики. А це можливо лише тоді, коли синтаксис є видимим, зрозумілим, прозорим («білим»). Звідси випливає, що мова є множиною (а не абстрактною сукупністю) певних елементів (речень, слів). Саме елементи множини тлумачаться як «білі

скриньки». Отже, мова розглядається на конкретно-структурованому рівні, який позначимо L.S (Language is a Set). Тлумачення мови як множини речень є сильною абстракцією, бо відкидаються зв'язки між реченнями, а самі речення розглядаються як незалежні одне від одного.

Принцип теоретико-множинного тлумачення формальної мови: на найвищому рівні абстракції синтаксичний аспект програм подається у вигляді формальної мови, яка тлумачиться як множини речень.

Відзначимо, що цей принцип намічає певну відмінність між семантикою, яка тлумачиться у теоретико-функціональному сенсі, та синтаксисом, для якого більш адекватною є теоретико-множинна формалізація.

Отже, перша над-абстрактна модель мови – це деяка множина $L = \{s_i \mid i \in I\}$. На цьому рівні абстракції нічого крім теоретико-множинних понять немає, тому немає і власне теорії мов (а є теорія множин). Треба переходити на інший рівень абстракції, який має хоч би деякі специфічні характеристики мов. Цей перехід, як відомо з діалектичної логіки, є переходом від цілого до частин. Тут головне – виділити структуру, що пов'язує ціле з частинами. Щоб це зробити, треба конкретизувати структуру речень.

Оскільки ми абстрагувались від семантики (зокрема від функцій та композицій), то приходимо до висновку, що частини речення не мають ніякої специфіки і тому вони не типізовані. Це означає, що вони належать одному класу і розглядаються як рівноправні. Цей клас називають алфавітом, а його елементи – літерами (символами). Тут спостерігається деяке розходження у вживаній термінології, бо у природних мовах речення складаються із слів, а вже слова – із літер. Але не забуваємо, що відбулося абстрагування від семантики, і що ми розглядаємо формальні мови, тому не слід навантажувати вживані терміни додатковим семантичним смислом.

Яка ж структура пов'язує літери в речення? Оскільки ніякої додаткової інформації не маємо, то згідно принципу розвитку від абстрактного до конкретного, обираємо найбільш просту, абстрактну структуру. Це є структура послідовності.

Абстрагуючись від лінгвістичних конотацій та асоціацій будемо для таких послідовностей вживати терміни «ланцюжок», «слово» або «речення». Відзначимо, що в комп'ютерній термінології часто вживається термін «слово» (word), в лінгвістичній – «речення» (sentence).

Структура послідовності використовується і в природних мовах, наприклад, друкування тексту на клавіатурі породжує саме послідовність літер. Крім того, усні розмови фактично визначають послідовності фонем.

Принцип тлумачення речень мови як послідовностей: на наступному рівні абстракції речення розглядаються як послідовності літер певного алфавіту.

Відзначимо, що тут абстрагуємось від порядку побудови ланцюжків. Наприклад, можна було б розрізнити ланцюжки (ab)c та a(bc). Але ми абстрагуємось від

генетичного аспекту (аспекту породження). Це подібно до того, як ми сприймаємо, наприклад, паркан, або ряд крісел в театрі. Для нас це послідовність секцій, і не важливо, в якому порядку вони зводилися. Цей рівень розгляду будемо позначати як L.S.S (Language is a Set of Sequences).

Наведені міркування та постулати дозволяють ввести наступну математичну модель формальної мови.

Друга, абстрактна модель формальної мови – це деяка підмножина $L \subseteq A^*$, де A^* – скінченні послідовності літер з алфавіту A . Це фактично є певним абстрактним поняттям формальної мови.

Зв'язок моделей задається операцією абстрагування $absL$, яка є операцією переведення множини в множину з інкапсуляцією структури послідовності речень.

Що можна досліджувати на такому рівні абстракції? В першу чергу, це теоретико-множинні операції та операції над послідовностями. Крім того, не слід забувати про операції, які зв'язують послідовності з множинами. Ці операції будуть описані пізніше у математичній частині розділу. Разом із тим такі операції занадто бідні і не розкривають особливостей синтаксичного аспекту. Викликано це тим, що ми абстрагувалися від способу, яким задається мова. І якщо для перших двох моделей таке абстрагування від генетичного та дескриптивного аспектів було корисним, бо дозволяє вивчати операції над мовами, то тепер потрібно ввести механізми породження мов (дескрипції мов).

З'являється третя модель формальної мови. Ця модель успадковує властивості першої моделі з додаванням нової характеристики – механізму породження мови.

Такий механізм можна називати породжуючою дескриптивною системою мови. Дуальний до породження механізм – сприйняття мов. Такий розподіл є традиційним для лінгвістики, де виділяють того, хто говорить (породження), і того, хто слухає (сприйняття). Такі ж механізми є і для програм: програмісти породжують програми, виконавці (зокрема, комп'ютери) – їх сприймають.

Отже, з'являється новий елемент поняття формальної мови – мовна дескриптивна система. Знову-таки, для визначення цієї системи будемо рухатись від абстрактного до конкретного.

Вважаючи, що породження є більш вагомим (ведучим) аспектом порівняно з сприйняттям, спочатку зосередимось на породженні.

Інтуїтивне розуміння породження полягає в тому, що це є процес, який веде від певних початкових елементів до заключних елементів (остаточно породжених елементів). Основні питання для уточнення цієї моделі полягають у наступному:

1. Елементи якої множини використовуються в процесі породження?
2. Які елементи є початковими?
3. Які елементи є заключними?
4. Як саме задаються операції породження одних елементів з інших?

Будемо вважати, що в процесі породження відбувається створення (перетворення) елементів множини St , початкові елементи задаються підмножиною I , а заключні – підмножиною F з St . Залишається питання про способи породження нових елементів. Найпростішим (найабстрактнішим) є випадок, коли породження

задається бінарним відношенням перетворень (переходів, транзицій) $tr \subseteq St \times St$.

Таким чином, отримали першу над-абстрактну модель породжуючої системи TS , яка задається параметрами (St, I, F, tr) . Така система породжує множину елементів $L(TS)$

наступним чином: береться елемент a з I , далі береться елемент b , такий що (a, b) належить tr , потім цей процес повторюється, аж поки не буде отриманий елемент c з

F . Цей елемент i входить до $L(TS)$. Тут фактично описаний процес побудови

рефлексивного транзитивного замикання tr^* відношення tr і вибору тих пар, перший елемент яких належить I , а другий - F . Це дозволяє записати формулу $L(TS) = \{ c \mid$

$(a, c) \in tr^*, a \in I, c \in F \}$.

Наведену модель у літературі часто називають транзиційною системою.

Зафіксуємо цей розгляд за допомогою наступного принципу.

Принцип розгляду механізму породження мови як транзиційної системи: над-абстрактну модель породжуючої системи тлумачимо як транзиційну систему

$TS = (St, I, F, tr)$ з визначеними раніше параметрами та яка визначає множину

породжуваних елементів мови наступною формулою: $L(TS) = \{ c \mid (a, c) \in tr^*, a \in I, c \in F \}$.

Зауважимо, що транзиційні системи можна використовувати і як моделі

сприйняття. В цьому випадку можна застосувати формулу: $L(TS) = \{ a \mid (a, c) \in tr^*, a \in I, c \in F \}$.

Як бачимо, різниця в породженні та сприйнятті полягає в тому, що в першому випадку елементи мови є результатами (вихідними даними) процесу породження, а в другому - аргументами (вхідними даними) процесу сприйняття. Крім того, можна говорити про те, що процес породження веде від позначення класу речень до окремого (індивідуального) речення, а процес сприйняття - навпаки, веде від індивідуального речення до позначення класу речень.

Транзиційні системи є моделями абстрактних динамічних систем, але суто мовні характеристики в них відсутні. Дійсно, як видно з наведеного визначення, ця над-абстрактна модель (транзиційна система) має той же рівень тлумачення елементів

мови як і над-абстрактна модель формальної мови. Треба тепер конкретизувати цю модель, щоб перейти до тлумачення мови як множини ланцюжків, тобто перейти на рівень абстрактної моделі мови. Будемо вважати, що система має породити мову $L \subseteq A^*$.

Розумно припустити, щоб множина St також конкретизувалась як множина послідовностей над певним алфавітом V . Таке тлумачення пов'язане з тим, що на

цьому рівні абстракції ми і не маємо інших структур речень. Зрозуміло, що $A \subseteq V$. Але

залишається питання, чи можна вважати, що $A=V$. Недовгі роздуми на цю тему

приводять до висновку, що брати випадок $A=V$ недоцільно, хоча в літературі такі системи досліджуються під назвою систем Tue . Справа в тому, що породження

вимагає певної допоміжної інформації, тому краще взяти B , що складається з двох частин: алфавіту A , та деякого допоміжного алфавіту N . Елементи алфавіту N дійсно

використовуються як допоміжні, тому вони не входять до ланцюжків породжуваної мови. За це їх називають нетермінальними символами (літерами), а елементи A

входять у заключні ланцюжки, тому їх називають термінальними символами. Таким чином, проведена наступна конкретизація: $St = (AUN)^*$, $I \subseteq$

$(AUN)^*$, $F \subseteq A^*$. Ця

конкретизація не є повною, тому що відношення tr залишається абстрактним, вона не враховує нову структуру St .

Щоб наступна конкретизація не була зайвою конкретизацією, треба «розумно»

скористатися тим, що St є множиною ланцюжків. Тому нова конкретизація tr

заснована на тому, що перетворення задаються не «глобально» на всьому ланцюжку, а «локально», лише на деякій частині ланцюжка. Інакше кажучи, від трансформації об'єкта як цілості переходимо до трансформації окремих частин. Наприклад, змії

заміняють свою «одежу» - шкіру - повністю, люди - частинами (можна замінити

окремо пальто, або сорочку і т.ін.). Аналогічно, під час ремонту складної техніки

можна замінити агрегат повністю, наприклад, замість пошкодженого двигуна

поставити новий, або замінити лише пошкоджену частину двигуна на справну.

При такому тлумаченні відношення переходів задається множиною локальних

правил P , а операція «глобалізації», яка одночасно є операцією абстракції $absG$,

дозволяє перетворити P в tr . Вкажемо наступну формулу: $abs(P) = \{ \lambda\alpha\beta \mid (\alpha, \beta) \in P, \}$

G

$\lambda, \rho \in (AUN)^*$. Деталі визначення abs буде наведено пізніше. У якості початкових

G

ланцюжків береться лише один певний символ S з N . Такий початковий символ

називають аксіомою (геном, «словоматкою»). Цей символ походить від терміну

«Sentence». Термінальний алфавіт у подальшому будемо позначати T (від терміну «Terminal»).

Таким чином, отримали другу абстрактну модель породжуючої системи, яка

задається параметрами $G = (N, T, P, S)$ і яка успадковує над-абстрактну модель

$T = (St, I, F, tr)$ наступним чином: $St = (TUN)^*$, $I = \{S\}$, $F = T^*$, $tr = abs(P)$.

Отриману модель

G

називають породжуючою граматику. Це відповідає традиційному вживанню

терміну «граматика», який означає сукупність правил побудови правильних речень мови. Зафіксуємо цей розгляд у вигляді відповідного принципу.

Принцип подання породжуючих систем породжуючими граматиками:

Абстрактну модель породжуючої системи тлумачимо як породжуючу граматику $G = (N, T, P, S)$ з наведеними вище параметрами, що є конкретизацією транзитивної

системи $TS = (St, I, F, tr)$ та визначає формальну мову $L(G)$ вказаним вище чином.

Чотири побудованих моделі та їх характеристики подано на рис. 4.2.

Малюнок подає взаємозв'язок моделей формальних мов. Він демонструє, що визначення формальної мови має щонайменше два виміри: перший рівень (вертикальний) задає рівень абстракції складових, другий рівень вводить нову дескриптивну складову.

Розрізнення рівнів абстракції визначень формальної мови та породжуючих граматик дозволяє досліджувати їх властивості на більш адекватному (більш абстрактному) рівні, де такі дослідження будуть простішими. Далі певні властивості переносяться (успадковуються) на більш конкретні рівні. Наприклад, теоретико-множинні властивості верхнього рівня (порожність мови, скінченність і таке інше) переносяться на більш конкретні рівні.

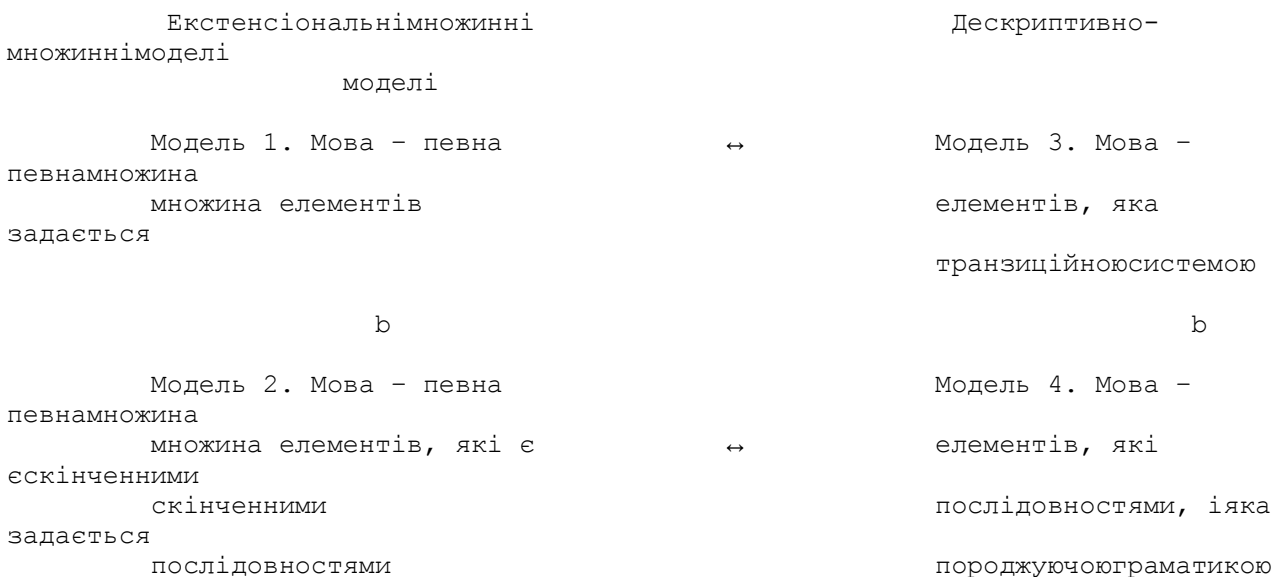


Рисунок 4.2. Моделі мов та граматик різного рівня абстракції

Зауваження 4.1. Як вже зазначалося, в літературі часто вживаються різні терміни для позначення одних і тих же понять. Наприклад, елемент формальної мови називають «реченням», «словом», «ланцюжком». Такий різнобій у термінології викликаний щонайменше двома обставинами:

а) дослідження формальних мов ведуться представниками різних наукових

шкіл, зокрема лінгвістики та інформатики. Термінологія таких шкіл

відрізняється одна від одної;

б) самі формальні мови досліджуються на різних рівнях абстракції, і той термін,

що є доречним на одному рівні абстракції, часто не є

відповідним (адекватним) іншому рівню абстракції. Наприклад, термін «слово» досить

адекватно відповідає рівню формальних мов, коли їх елементи задаються як послідовності символів, разом з тим цей термін не зовсім адекватний поданню елементів мови за допомогою граматик, бо текст великої програми, що складається з багатьох синтаксичних категорій, неадекватно тлумачити як одне слово.

Ми і в тексті цього посібника не будемо занадто вимогливими і дозволимо собі вживати різні терміни для одних і тих же понять.

Зробимо також декілька методологічних зауважень.

1. Про опосередкування. Вивчення мови відбувається не безпосередньо (як множини ланцюжків), а опосередковується механізмом дескриптивних систем. Тобто вивчаються не стільки мови, скільки граматики. У книжках цей факт явно не акцентується. Це подібно до того, як вивчення натуральних чисел опосередковується вивченням операцій та відношень, заданих на натуральних числах. Так само і вивчення програм фактично опосередковується вивченням їх композицій.

2. Про визначення породжуючої граматики. В існуючій літературі формальна граMATика визначається як четвірка $G=(N, T, P, S)$ з визначеними раніше параметрами. На наш погляд, таке визначення не є вдалим, оскільки тут задаються лише індивідуальні параметри (екстенсіональні аспекти) і зовсім не розкриваються механізми породження мов (інтенсіональні аспекти), які визнаються значно пізніше. Те саме часто можна зустріти в деяких книжках з теорії алгоритмів, дискретної математики, логіки. Визначення граматики повинно починатися з властивостей класу породжуючих граматик, і лише потім потрібно задавати конкретні (індивідуальні) параметри.

3. Про відношення вивідності. Воно задається через єдність загального та одиничного. Це подібно, наприклад, до інструкції з користування телевізором. Механізм опису апелює не до конкретного телевізора в вашій кімнаті, а до будь-якого телевізора відповідної моделі. Тобто інструкція спирається на загальні параметри, які потім інтерпретуються для кожного конкретного телевізора. Це досить тонке питання, і його слід розуміти.

Таким чином, розглянуто моделі формальних мов різного рівня абстракції. Ці моделі можна назвати інтенсіональними моделями, бо вони характеризують властивості визначення формальних мов. Для подання конкретних мов використовують екстенсіональні моделі. В наступних розділах будемо спиратися на екстенсіональні визначення мов. Зазначимо, що традиційно обмежуються лише екстенсіональними визначеннями, не розкриваючи інтенсіональних.

Сформулюємо основні висновки нашого аналізу.

1. Поняття формальної породжуючої граматики має два аспекти:
інтенціональний та екстенціональний.
2. Екстенціональний аспект більш простіший і задається як клас об'єктів,
кожен з яких є четвіркою виду $G=(N, T, P, S)$, де N і T – скінчені алфавіти,
*
 $N \cap T = \emptyset, P \subseteq (N \cup T)^* \times N \times (N \cup T)^* \times (N \cup T)^*$, P скінчена і $S \in N$.
3. Іntenціонал поняття породжуючої граматики полягає в наступному:
кожна граMATика призначена для породження формальної мови,
породження відбувається за допомогою відношення безпосереднього виводу, а мова – це множина термінальних ланцюжків, що виводяться з аксіоми за скінченну кількість кроків.
4. Поняття формальної граматики є єдністю обох аспектів, тут важливо підкреслити, що інтенціональний аспект є тим загальним, що може використовуватись у конкретних (індивідуальних, одиничних) граMATиках, бо задається фактично схема визначень, яка інтерпретується на конкретних значеннях параметрів.

Зроблений методологічний та інтенціональний аналіз поняття формальної мови дозволяє тепер перейти до введення екстенціональних визначень формальних мов та граMATик.

4.2. Визначення основних понять формальних мов

Визначення 4.1. Алфавітом називають скінченну непорожню множину символів (літер). Позначатимемо алфавіт знаком Σ .

Приклад 4.1. Найчастіше використовуються наступні алфавіти:

1. $\Sigma = \{0,1\}$ – бінарний чи двійковий алфавіт;
2. $\Sigma = \{a, b \dots z\}$ – множина літер англійського алфавіту.

Визначення 4.2. Ланцюжком, чи інколи словом, реченням, рядком, в алфавіті Σ називають скінченну послідовність символів з Σ .

Приклад 4.2. Послідовність 01101 – це ланцюжок у бінарному алфавіті $\Sigma = \{0,1\}$.

Ланцюжок 111 також є ланцюжком у цьому алфавіті.

Визначення 4.3. Порожній ланцюжок – це ланцюжок, який не містить жодного символу. Цей ланцюжок позначається ε . Його можна розглядати як ланцюжок у довільному алфавіті.

Визначення 4.4. Довжина слова (послідовності) w позначається $|w|$; якщо $\Sigma' \subseteq \Sigma$, то довжина послідовності, утвореної з w видаленням тих символів, що не належать Σ' , позначається $|w|_{\Sigma'}$; якщо $a \in \Sigma$, то $|w|_{\Sigma'}$ означає $|w|$ і задає кількість входжень

$$\Sigma' \quad a \quad \{a\}$$

символу a в w .

Приклад 4.3. Для ланцюжка abcбасаасссб маємо:

$$|abcбасаасссб| = 12, |abcбасаасссб|_{\{a, c\}} = 9, |abcбасаасссб|_{\{a, c\}} = 5.$$

Визначення 4.5. Якщо w та u - ланцюжки в алфавіті Σ , то ланцюжок wu (результат дописування слова u в кінець слова w) називається конкатенацією

(конкатенацією, зчепленням) слів w та u . Іноді конкатенацію слів позначають $w \cdot u$.

Визначення 4.6. Якщо w - ланцюжок в алфавіті Σ , то ланцюжок $w \cdot w \cdot \dots \cdot w$

14243

n раз

називається n -ю степенню w і позначається w^n .

За визначенням, $w^0 = \varepsilon$.
 Приклад 4.4. $a^3 = aaa$, $(ab)^3 = ababab$,
 $(abc)^2 = abcabc$

Визначення 4.7. Множина всіх ланцюжків в алфавіті Σ позначається Σ^* і називається вільною напівгрупою, породженою Σ . Множина всіх непорожніх

ланцюжків позначається Σ^+ .

Приклад 4.5. Якщо $\Sigma = \{a\}$, то $\Sigma^* = \{\varepsilon, a, aa, aaa, \dots\}$, $\Sigma^+ = \{a, aa, aaa, \dots\}$.

Термін «вільна напівгрупа» веде своє походження з алгебри. Напівгрупою

називається множина з асоціативною бінарною операцією. Якщо така множина має одиничний елемент, то її називають моноїдом. Напівгрупа вільна, якщо ніяких інших

співвідношень (крім асоціативності) немає. В теорії формальних мов часто

вважають, що напівгрупа має одиничний елемент. Множину Σ^* можна розглядати як вільну напівгрупу з одиницею. Конкатенація є асоціативною операцією, а порожній ланцюжок ε є одиницею, тому що для довільного ланцюжка w маємо $w \cdot \varepsilon = \varepsilon \cdot w = w$.

Такий розгляд множини ланцюжків Σ^* дозволяє перейти до більш багатой структури напівгрупи та використати алгебраїчні властивості для дослідження цієї множини.

Іншою обставиною є те, що подаючи множину послідовностей у вигляді напівгрупи, ми ототожнюємо символ з послідовністю довжини 1, що складається з цього символу. Тому замість двохосновної моделі (алфавіт та множина послідовностей), яка розрізняє символи та послідовності, розглядається лише одна основа - множина ланцюжків. Це спрощує дослідження.

Визначення 4.8. Ланцюжок t є підланцюжком ланцюжка w , якщо $w = utv$ для деяких ланцюжків u та v .

Нарешті, введемо поняття формальної мови.

*

Визначення 4.9. Якщо $L \subseteq \Sigma^*$, то L називається формальною мовою (або просто мовою) над алфавітом Σ (в алфавіті Σ).

n n n

2 2 2 3 3 3

Приклад 4.6. Множина ланцюжків $\{a b c \mid n \geq 0\} = \{\epsilon, abc, a b c, a b c, \dots\}$ є формальною мовою над алфавітом $\{a, b, c\}$.

Зауважимо, якщо L є мовою над Σ , то можна стверджувати, що L - це мова над будь-яким алфавітом Σ' , що містить Σ . Інакше кажучи, є певна консервативність поняття мови до розширення алфавіту. Крім того, є монотонність множини мов відносно розширення алфавіту, бо це веде до збільшення множини мов.

4.3. Операції над формальними мовами

Операції над формальними мовами розподіляються на два класи. Перший клас операцій відповідає над-абстрактній моделі мов, у якій вони мають інтенціонал множини. Тому всі теоретико-множинні операції можна застосовувати для мов. У

першу чергу, це операції об'єднання \cup , перетину \cap та різниці \setminus . Вважаємо, що мови

- аргументи цих операцій - задані над одним і тим самим алфавітом. Якщо це не так, то будемо новий алфавіт, який є об'єднанням алфавітів мов-аргументів. Тоді всі мови-аргументи будуть мовами над одним алфавітом. Якщо мова L є мовою в алфавіті Σ , то мова $\Sigma^* \setminus L$ називається доповненням мови L відносно алфавіту Σ . Коли з контексту ясно, про який алфавіт йде мова, говорять просто, що мова $\Sigma^* \setminus L$ є

доповненням мови L і позначається \bar{L} .

Приклад 4.7. Нехай $L_1 = \{a^n b^m \mid n, m \geq 0\}$, $L_2 = \{a^n b^m c^m \mid n, m \geq 0\}$. Тоді

$$L_1 \cap L_2 = \{a^n b^m c^m \mid n \geq 0, m \geq 0\}, L_1 \setminus L_2 = \{a^n b^m \mid n \geq 0, m \geq 0, m \neq 0\}.$$

Другий клас операцій над формальними мовами відповідає абстрактній моделі мов, коли їх елементи тлумачаться як послідовності символів. Серед цих операцій, у першу чергу, відзначимо операцію добутку (конкатенації) мов, яка є похідною від операції конкатенації ланцюжків. Для позначення цієї операції використовуємо той самий символ операції конкатенації.

Визначення 4.10. Нехай $L_1, L_2 \subseteq \Sigma^*$. Тоді $L_1 \cdot L_2 = \{xy \mid x \in L_1, y \in L_2\}$. Мова $L_1 \cdot L_2$ називається конкатенацією мов L_1 та L_2 .

$$L_1 \cdot L_2 = \{xy \mid x \in L_1, y \in L_2\}$$

Приклад 4.8. Якщо $L_1 = \{a, abac\}$ та $L_2 = \{bcc, c\}$, то $L_1 \cdot L_2 = \{ac, abacb, abacbc, abcc\}$.

Найважливішою характеристикою операції добутку мов є її асоціативність. Одиницею цієї операції є мова, що складається з порожнього ланцюжка, тобто мова

{ ϵ }. Похідною від добутку є операція піднесення до степені.
 Вважаємо, що

$$L^n = L \cdot \dots \cdot L$$
 n раз

Маючи степінь, дамо індуктивне визначення ітерації, яку ще називають замиканням Кліні, або «зірочкою Кліні».

n

Визначення 4.11. Ітерацією мови L (позначається L^*) називається мова L^* .

n \in

Приклад 4.9. $\{ab\}^* = \{\epsilon, ab, abab, ababab, \dots\}$.

Для мови, що складається з одного символу, фігурні дужки часто опускають і пишуть, наприклад, a^* замість $\{a\}^*$.

Визначення 4.12. Оберненням або дзеркальним образом ланцюжка w

(позначається w^R) називається ланцюжок, складений із символів w, взятих в оберненому порядку.

Приклад 4.10. Якщо $w = abc$, то $w^R = cba$.

Визначення 4.13. Нехай $L \subseteq \Sigma^*$. Тоді $LR = \{w^R \mid w \in L\}$.

Крім вищенаведених операцій можна ввести й інші операції цього рівня, наприклад, ділення мов, проекції на під алфавіт тощо. Також можна ввести операції

індуктивного та рекурсивного визначення мов. Такі операції будуть розглянуті пізніше.

Хоч і ми визначили чимало операцій над формальними мовами, їх недостатньо, щоб задавати важливі типи мов. Тому потрібно переходити до методів дескриптивного подання мов, тобто до граматики.

За способом подання правильних ланцюжків формальні граматики поділяються на породжуючі і розпізнаючі (граматики породження та сприйняття). До породжуючих граматики відносяться граматики, які дозволяють побудувати будь-який правильний ланцюжок з зазначенням його структури і не дозволяють побудувати жодного неправильного ланцюжка. Розпізнаюча граMATика - це граMATика, яка дозволяє визначити, чи є довільно обраний ланцюжок правильним і, якщо він є правильним, визначити його структуру.

Формальні граматики широко застосовуються в лінгвістиці, інформатиці, логіці,

та програмуванні у зв'язку з вивченням природних та штучних мов.

Почнемо з розгляду породжуючих граматики.

4.4. Породжуючі граматики

Як зазначалося на початку розділу, породжуючі граматики можуть розглядатися як конкретизації транзиційних систем або дедуктивних систем. Для таких систем

головним є відношення переходів. У граматиках відношення переходу (виведення)

задається за допомогою правил граматики (продукцій), які мають вигляд $\alpha \rightarrow \beta$, де α та β – ланцюжки в певному алфавіті Σ . Таке правило дозволяє перетворити

ланцюжок $\gamma_1 \gamma_2 \dots \gamma_n$ у ланцюжок $\gamma'_1 \gamma'_2 \dots \gamma'_n$ ($\gamma_i, \gamma'_i \in \Sigma^*$) тоді і тільки тоді, коли $\gamma_i = \delta_i \alpha_i$, $\gamma'_i = \delta_i \beta_i$

для деяких ланцюжків δ_i і α_i, β_i , що належать Σ^* . Змістовно це правило дозволяє

замінити підланцюжок γ_1 , який співпадає з лівою частиною правила, на праву його частину, отримуючи ланцюжок γ' .

Нехай P – множина продукцій. Тоді кожна продукція з P може розглядатися як правило виводу на Σ^* . Сама послідовність правил, використаних в процесі породження деякого ланцюжка, є його виводом. Визначена таким способом граMATика представляє собою формальну систему. Відомими прикладами формальних систем слугують логічні числення (числення висловлювань, числення предикатів), які детально вивчаються у відповідних розділах математичної логіки.

Низку формальних систем для програм буде наведено в цьому посібнику.

Наведені міркування приводять до наступного визначення.

Визначення 4.14. Породжуючою граMATикою (граMATикою типу 0) називається четвірка $G = (N, T, P, S)$, де N і T – скінчені алфавіти, $N \cap T = \emptyset$,

$P \subseteq (N \cup T)^* \times N \times (N \cup T)^*$, P скінчена і $S \in N$. Тут

- N – нетермінальний алфавіт (допоміжний алфавіт), його елементи

називаються нетермінальними символами, нетерміналами, змінними;

- T – термінальний алфавіт (основний алфавіт), його елементи називаються

термінальними символами або терміналами;

- P – множина продукцій. Продукцію $(\alpha, \beta) \in P$ інколи називають правилом підстановки, правилом виводу, або просто правилом і записують у вигляді

$\alpha \rightarrow \beta$;

- S – початковий символ (аксіома).

Зробимо декілька зауважень до наведеного визначення. По-перше, слід пам'ятати про те, що вказана четвірка є просто екстенсіональним (одиничним) об'єктом

поняття породжуючої граматики і без посилань на інтенціональні властивості граматики вона не є самодостатньою. Оскільки для нас важливою є єдність екстенціональних та інтенціональних аспектів, то ми вважаємо, що і у визначенні граматики така єдність повинна бути відзначена. По-друге, умова, що ліва частина правил належить множині $(N \cup T)^* N (N \cup T)^*$ означає, що в лівій частині обов'язково повинен бути нетермінал. Це обмеження фактично індукується

інтенціональною ознакою породження, бо воно не повинно вестися із завершеного (термінального) ланцюжка. Разом з тим у лівій частині можуть фігурувати і термінальні символи, які можуть замінюватись на символи правої частини правила. Це дозволяє говорити, що визначення породжуючої граматики не достатньо підтримує розподіл символів на термінальні та нетермінальні. Як вже зазначалось, від такого часткового рішення можна відмовитись, розглядаючи системи T_u (де нема розподілу на термінальні та нетермінальні символи), або розглядаючи контекстні граматики (де дозволена заміна лише нетермінального символу, але в певному контексті). По-третє, множини T, N, P були визначені як скінченні. Проте, більшість результатів, що стосуються породжуючих граматики, може бути перенесена на нескінченні множини.

У прикладах нетермінальні символи зазвичай позначаємо великими літерами, термінальні - маленькими, для правил з однаковими лівими частинами $\alpha \rightarrow \beta, \dots, \alpha \rightarrow \beta$ часто будемо використовувати скорочений запис $\alpha \rightarrow |\beta \dots |$ β .

Крім того, у прикладах інколи будемо задавати граматику не як відповідну четвірку, а просто списком правил, вважаючи, що алфавіт N складають всі великі літери, які зустрічаються в правилах, а алфавіт T - всі маленькі літери, що зустрічаються в правилах. При цьому ліва частина першого правила є початковий символ S .

Повторимо ще раз визначення відношення безпосередньої вивідності, яке задає граматику $G=(N, T, P, S)$.

Визначення 4.15. Нехай задано граматику $G=(N, T, P, S)$. Пишемо $\gamma \Rightarrow \gamma$

$$1 \quad G \quad 2 \quad 1 \quad 2$$

*

є $(N \cup T)$, якщо $\gamma = \delta \alpha \delta$, $\gamma = \delta \beta \delta$ для деяких слів δ , $\delta \in (N \cup T)$, і $\alpha \rightarrow \beta \in P$.

Коли з контексту зрозуміло, про яку граматику йде мова, замість \Rightarrow_G можна

писати просто \Rightarrow .

Визначення 4.16. Рефлексивне транзитивне замикання відношення безпосередньої вивідності називається відношенням вивідності і позначається через

*

\Rightarrow^*G (або $\rightarrow G$).

Нагадуємо, що рефлексивним транзитивним замиканням бінарного відношення R на множині ST є найменше відношення R' , яке містить R , і яке є рефлексивним та транзитивним, тобто

- 1) $sR's$ для всіх $s \in ST$;
- 2) якщо $s_1 R_2 s_2$ та $s_2 R_3 s_3$, то $s_1 R_3 s_3$.

Наведене визначення не вказує на спосіб побудови рефлексивного транзитивного замикання. Разом з тим з властивостей рефлексивного транзитивного замикання

впливає, що ланцюжок γ виводиться з ланцюжка γ ($\gamma \Rightarrow^* \gamma$) тоді і тільки тоді,

$G \vdash \gamma$ n 0 0

коли деяка послідовність (можливо порожня) замінив лівих частин продукцій з P їхніми правими частинами переводить ланцюжок γ в γ . Інакше кажучи, коли існує

$0 \quad n$

послідовність виду $\gamma \Rightarrow \gamma \Rightarrow \dots \Rightarrow \gamma$ ($n \geq 0$).

$0 \quad G \quad 1 \quad G \quad G \quad n$

Визначення 4.17. Послідовність ланцюжків $\gamma_1, \gamma_2, \dots, \gamma_n$, така, що $\gamma_i \Rightarrow \gamma_{i+1}$ для

$0 \leq i < n$

$1 \leq i \leq n$, називається виводом (виведенням) γ_n з γ_0 в G . Число n називається довжиною

$n \quad 0$

(кількістю кроків) цього виведення.

Зауважимо, що для довільного ланцюжка γ має місце $\gamma \Rightarrow^* \gamma$ (адже можливе

G

виведення довжини 0). Слід також сказати, що вивід $\gamma_0, \gamma_1, \dots, \gamma_n$ не дає однозначного

$0 \quad 1$

n

опису, які саме правила граматики були застосовані і до яких підланцюжків. Якщо така інформація є важливою, то можна застосовувати розмічений вивід, вказуючи входження лівої частини правила, що замінюється, та позначення правила, що було застосовано.

Визначення 4.18. Ланцюжки, що виводяться з певного нетерміналу A ,

називаються його словоформами (А-словоформами), або його сентенційними формами. Якщо вивід іде із аксіоми, то говоримо просто - словоформа, або сентенційна форма.

Приклад 4.11. Для граматики з правилами $\{ S \rightarrow aBSc, S \rightarrow \varepsilon, B \rightarrow \varepsilon \}$ та аксіомою S словоформами будуть ланцюжки aBSc, aBaBaBSccc, aaaBccc, та інші, які виводяться з S.

Центральним є визначення, яке задає мову, що породжується граматикою.

Визначення 4.19. Мова, що породжується граматикою G, - це множина ланцюжків $L(G) = \{ w \mid S \Rightarrow^* w, w \in T^* \}$. Будемо також говорити, що граматика G

G

породжує мову L(G).

Говорячи просто, мова L(G) є мовою, що породжується граматикою G, якщо вона складається із ланцюжків (слів) у термінальному алфавіті, які виводяться із аксіоми S.

Інтенціональні аспекти цього визначення були обговорені раніше. Зауважимо, що в попередньому підрозділі для побудови моделей формальних мов використовувався підхід від загального (абстрактного) до конкретного (одиночного). Спочатку

вводилось поняття мови як множини речень, а в кінці побудов вводилося поняття породжувачої граматики. Це був інтенціональний аналіз поняття граматики. В цьому підрозділі зроблено навпаки - ідуть екстенціональні побудови: від екстенціонального визначення породжувачої граматики через поняття виводу та його абстракції і завершується поняттям мови, що породжується граматикою. Тут фактично можна говорити про визначення знизу-вверх: від індивідуальних понять - до загальних.

Ці міркування говорять про те, що поняття породжувачої граматики треба розглядати як єдність екстенціоналу, що визначається четвірками вигляду $G = (N, T, P, S)$ з указаними раніше параметрами, та інтенціоналу, що дає загальне визначення

відношення безпосередньої вивідності \Rightarrow , його рефлексивного транзитивного замикання \Rightarrow^* , та визначення мови, що породжується, за формулою $L(G) = \{ w \mid S \Rightarrow^* w, w \in T^* \}$.

G

4.5. Приклад породжувачої граматики та її властивості

Розглянемо на прикладі основні визначення та методи доведення властивостей граматик та породжуваних мов.

Нехай задана граматика $G = (\{S, B\}, \{a, b, c\}, P, S)$, де

$P = \{$

$P1: S \rightarrow aBSc,$

$P2: S \rightarrow \varepsilon,$

$P3: Ba \rightarrow aB,$

$P4: Bb \rightarrow bB,$

$P5: Bc \rightarrow bc$

$\}$

Тут P1, P2, P3, P4, P5 - мітки відповідних правил.

Щоб зрозуміти, яку мову породжує ця граматики, побудуємо декілька виводів.

Будемо використовувати розмічені виводи. Маємо:

P 2

1. $S \Rightarrow \epsilon$.

P 1 P 2 P 5

2. $S \Rightarrow aBSc \Rightarrow aBc \Rightarrow abc$.

P 5 P 1 P 1 P 2 P 3 P 5 P 4

3. $S \Rightarrow aBSc \Rightarrow aBaBSc \Rightarrow aBaBcc \Rightarrow aaBBcc \Rightarrow aaBbcc \Rightarrow aabBcc \Rightarrow aabbcc$.

P 3 P 1 P 1 P 1 P 3

P 3

4. $S \Rightarrow aBSc \Rightarrow aBaBSc \Rightarrow aBaBaBSc \Rightarrow aBaBaBSc \Rightarrow aBaaBBSc$
 $\Rightarrow aaBaBBSc \Rightarrow$

P 4 P 5 P 2 P 5 P 4

$aaaBBSc \Rightarrow aabBBSc \Rightarrow aabBBcc \Rightarrow aabBbcc \Rightarrow aabBbcc$
 $aaaBbcc \Rightarrow aabBBSc \Rightarrow$

P 4 P 5

$aaabBbcc \Rightarrow aaabbBcc \Rightarrow aaabbbcc$

Отже, в цих виводах породжено ланцюжки ϵ , abc , $aabbcc$, $aaabbbcc$. Це дозволяє

$n \ n \ n$

висунути припущення, що граматики G_3 породжує мову $L_3 = \{a^n b^n c^n \mid n \geq 0\}$.

Доведемо,

що це дійсно так.

Теорема 4.1. $L(G_3) = \{a^n b^n c^n \mid n \geq 0\}$.

$n \ n \ n$

Спочатку доведемо, що $L(G_3) \subseteq \{a^n b^n c^n \mid n \geq 0\}$. Це доведення базується на наступній

лемі, ідея якої полягає в тому, щоб сформулювати загальний вигляд ланцюжків в

об'єднаному алфавіті, що виводяться в G_3 (загальний вигляд словоформ).

Лема 4.1. Нехай $S \Rightarrow^* G_3 \alpha$ ($\alpha \in (NUT)^*$). Тоді існує $n \geq 0$ таке, що

- 1) $\alpha = \beta \gamma \delta$, де $\beta \in \{a, B\}^*$, $\gamma = S$ або $\gamma \in \{b, B\}^*$;
 - 2) $|\alpha| = |\beta| + |\gamma| + |\delta| = n$.
- $a \qquad c \qquad \{b, B\}$

Доведення лемі: індукція по довжині виводу.

База індукції. Для виводу довжини 0 маємо, що $n=0$, $\alpha=S$. Значить, α має вигляд

$\beta\gamma\epsilon$ ($\beta=\epsilon$, $\gamma=S$, $\epsilon=\epsilon$) і тому $|\alpha| = |\alpha| = |\alpha| = 0$. Лема виконується.

Крок індукції. Нехай лема виконується для усіх виводів довжини $k \geq 0$. Доведемо, що вона виконується для виводів довжини $k+1$.

Візьмемо довільний вивід $S \Rightarrow \alpha \Rightarrow \dots \Rightarrow \alpha \Rightarrow \alpha$. Особливість виводів полягає

1 $G3$ 1 $G3$ $G3$ k $G3$ k+

в тому, що початкова послідовність $S \Rightarrow \alpha \Rightarrow \dots \Rightarrow \alpha$ буде виводом довжини k , і

k $G3$ 1 $G3$ $G3$

тому за індуктивним припущенням для α виконується твердження леми. Інакше

кажучи, є деяке $n \geq 0$, таке що $\alpha = \beta\gamma\epsilon$ та $|\alpha| = n$. Безпосередній вивід k $\{b, B\}$

$\alpha \Rightarrow \alpha$ здійснюється за допомогою одного з правил P1, P2, P3, P4, P5.

Розглянемо k $G3$ k+ 1

ці правила по черзі.

P 1

1. Нехай вивід $\alpha \Rightarrow \alpha$ відбувся із застосуванням правила P1, тобто $\alpha \Rightarrow \alpha$ k $G3$ k+ 1

Тоді α_k має вигляд $\beta S \epsilon_n$ ($\beta \in \{a, B\}^*$), а $\alpha_{k+1} = \beta a B S \epsilon_{n+1}$. Тому α_{k+1} має вигляд $\beta' S \epsilon_{n+1}$

($\beta' \in \{a, B\}^*$), як того вимагає лема, і крім того, $|\alpha| = |\alpha| = |\alpha| = n+1$.

k+ 1 a k+ 1 c k+ 1 $\{b, B\}$

Для цього випадку лему доведено.

P 2

2. Нехай вивід $\alpha \Rightarrow \alpha$ відбувся застосуванням правила P2, тобто $\alpha \Rightarrow \alpha$.

$k \quad G3 \quad k+1$

Тоді α_k має вигляд $\beta S c$ ($\beta \in \{a, B\}^*$), а $\alpha_{k+1} = \beta c$. Тому α_{k+1} можна подати у вигляді

$\beta \gamma c^n$ ($\gamma = \varepsilon$), як того вимагає лема, і крім того, $|\alpha_{k+1}| = |\alpha_k| = n$. Для

$\alpha_{k+1} a \quad \alpha_{k+1} c \quad \alpha_{k+1} \{b, B\}$

цього випадку лему доведено.

3. Нехай вивід $\alpha \Rightarrow \alpha$ відбувся застосуванням правила P3. Оскільки α має

загальний вигляд $\beta \gamma c$, то правило P3 може бути застосовано лише для

підланцюжка β . Це правило не змінює ані загального вигляду α_{k+1} , ані

кількості символів, тому $|\alpha_{k+1} a| = |\alpha_{k+1} c| = |\alpha_{k+1} \{b, B\}| = n$. Для цього випадку лему доведено.

4. Нехай вивід $\alpha \Rightarrow \alpha$ відбувся застосуванням правила P4. Оскільки α має

загальний вигляд $\beta \gamma c$, то правило P4 може бути застосоване лише для

підланцюжка γ (хоча можливо замінюване B є останнім символом β).

Застосування правила не змінює ані загального вигляду α_{k+1} , ані кількості

символів, тому $|\alpha_{k+1} a| = |\alpha_{k+1} c| = |\alpha_{k+1} \{b, B\}| = n$. Для цього випадку лему доведено.

5. При застосуванні правила P5 твердження леми також залишається справедливим. Лему доведено.



Нехай тепер $S \Rightarrow^* G3t$ ($t \in T^*$). Оскільки t - ланцюжок в термінальному алфавіті, то він

не містить нетерміналів, і за твердженням леми існує $n \geq 0$ таке, що $t = \beta \gamma c^n$, де $\beta \in a^*$,

$\gamma \in b^*$ та $t = a^n b c^n$. Звідси випливає, що $t = a^n b c^n$. Інакше кажучи, будь який $a \quad c \quad b$

термінальний ланцюжок, що виводиться в G_3 , належить мові L_3 , тобто $L(G_3) \subseteq L_3$.

Доведемо тепер зворотне включення, тобто $L_3 \subseteq L(G_3)$.

$n \ n \ n$

Для цього продемонструємо, як побудувати вивід ланцюжка $a^n b^n c^n$ для довільного

n . Використовуємо індукцію за n .

P 2

$n \ n \ n$

База індукції. Ланцюжок $a^n b^n c^n$ ($=\varepsilon$) отримуємо виводом $S \Rightarrow \varepsilon$.

$n \ n \ n$

$n \ n \ n$

Крок індукції. Нехай існує вивід слова $a^n b^n c^n$ ($n \geq 0$),

тобто $S \Rightarrow^* G_3 a^n b^n c^n$. Доведемо,

що існує вивід слова $a^{n+1} b^{n+1} c^{n+1}$, тобто $S \Rightarrow^* G_3 a^{n+1} b^{n+1} c^{n+1}$.

Відповідний вивід будемо наступним чином. Спочатку до аксіоми застосуємо

$n \ n \ n$

перше правило, а потім зробимо вивід слова $a^n b^n c^n$. Отримали наступний вивід:

P 1

$n \ n \ n$

$S \Rightarrow a^n B S c \Rightarrow^* G_3 a^n B a^n b^n c^n$. Далі n разів застосуємо правило P3. Отримали ланцюжок

$a^n a^n B^n b^n c^n$ ($= a^{2n} B^n b^n c^n$). Після цього n разів застосуємо правило P4. Отримали

ланцюжок $a^{n+1} b^{n+1} c^{n+1}$. Нарешті, застосуванням правила P5 отримуємо ланцюжок $a^{n+1} b^{n+1} c^{n+1}$.

Теорему 4.1 доведено. ■

Наступним кроком у вивченні породжувачих граматики буде їх класифікація.

4.6. Ієрархія граматики Хомського

Дослідження будь-якого класу предметів, як правило, починається з їх класифікації. Це методологічне зауваження стосується і граматики. Однією з перших була класифікація, запропонована Н. Хомським. Суть цієї класифікації полягає в поступових обмеженнях, що накладаються на праві та ліві частини продукцій.

Визначені Хомським чотири типи граматики виявились інваріантними відносно різних уточнень породжувачих граматики. Крім того, цим типам граматики природним чином відповідають типи автоматів, що розпізнають (сприймають) формальні мови.

Визначення 4.20.

• Граматиками типу 0 називають довільні породжувачі граматики загального

виду, що не мають жодних обмежень на правила виводу.

• Граматиками типу 1 (нескорочувачими граматиками) називають породжувачі

граматики, кожне правило яких має вигляд $\alpha \rightarrow \beta$, де $|\alpha| \leq |\beta|$
 $(\alpha \in (NUT)^* N (NUT)^* ,$
 $\beta \in (NUT)^+) .$

- Граматиками типу 2 (контекстно-вільними граматики, КВ-граматики) називають породжуючі граматики, кожне правило яких має вигляд $A \rightarrow \beta$, де $A \in N, \beta \in (NUT)^* .$
- Граматиками типу 3 (праволінійними граматики) називають породжуючі граматики, кожне правило яких має вигляд $A \rightarrow \alpha B$ або $A \rightarrow \alpha$, де $A, B \in N,$

$\alpha \in TU\{\epsilon\} .$ До граматик типу 3 відносять і ліволінійні граматики, кожне правило

яких має вигляд $A \rightarrow B\alpha$ або $A \rightarrow \alpha$, де $A, B \in N, \alpha \in TU\{\epsilon\} .$

Приклад 4.12. Граматика G_3 з попереднього підрозділу є граматикою типу 0.

Якщо з цієї граматики видалити правило P_2 , вона стане граматикою типу 1, бо не буде скорочуючих правил. Граматика з правилами $\{S \rightarrow AR, R \rightarrow bRc, R \rightarrow \epsilon, A \rightarrow aA, A \rightarrow \epsilon\}$ є контекстно вільною граматикою (типу 2), а її підграматика $\{A \rightarrow aA, A \rightarrow \epsilon\}$ є праволінійною граматикою (типу 3).

Кожному типу граматик відповідають мови, яким будемо приписувати той же тип, що має граматика, яка її породжує.

Тип 1

Тип 0

Тип 2

Тип 3

Рисунок 4.3. Співвідношення типів граматик

Error! Reference source not found. 4.3 демонструє співвідношення типів граматик. Як

бачимо, граматики типів 2 та 3 не є підкласами граматик типу 1. Це викликано тим,

що наведені класи граматик мають скорочуючі правила виду $A \rightarrow \epsilon$, які заборонені в граматики типу 1. Як буде показано далі, ці відмінності не дуже позначаються на

класах мов, породжуваних граматики типів 2 та 3, і якщо ігнорувати порожній ланцюжок, то класи мов типів 3 та 2 будуть підкласами мов типу 1.

Перш ніж переходити до вивчення класів мов, покажемо, що без втрати виразної сили можна розглядати більш обмежений клас граматик, які називаються загально-контекстними граматики. Цей клас видається більш природним для породження, ніж загальні породжуючі граматики.

Визначення 4.21

- Загально-контекстними називають породжуючі граматики, кожне правило яких має вигляд $k A_k \rightarrow k \beta_k$, де $A \in N, k_1, k_2, \beta \in (NUT)^* .$

Ланцюжки $k_1 \dots k_n$ $i_1 \dots i_m$

1	2	1	2	1	2
---	---	---	---	---	---

називають лівим та правим контекстом символу A у вказаному правилі.

- Контекстно-залежними називаються загально-контекстні граматики з

нескорочуваними правилами, тобто правила мають вид $\alpha \rightarrow \beta$, де $|\alpha| \geq 1$.

1 2 1 2

+

$\beta \in (NUT)$ (це означає, що $|\beta| \geq 1$).

Контекстно-залежні граматики інколи називають граmaticами безпосередньо складових (БС-граматиками).

Слід сказати, що назва контекстно-залежних граmatic не зовсім вдала, бо не підкреслює нескорочуваність правил граматики, а характеризує радше довільний клас контекстних граmatic. Це ж стосується і назви граmatic безпосередньо складових.

Граматики типів 2 та 3 (будучи безконтекстними) безпосередньо є підкласами загально-контекстних граmatic.

З визначень видно, що класи загально-контекстних та контекстно-залежних граmatic є власними підкласами відповідно граmatic типів 0 та 1. Разом з тим відповідні класи мов співпадають, тобто використання лише контекстних правил не обмежує породжувальну здатність граmatic.

Доведемо цей факт співпадіння мов.

Нехай $G = (N, T, P, S)$ - довільна породжувача граmatic типу 0. Побудуємо

0

еквівалентну їй загально-контекстну граmaticу. Алгоритм побудови наступний.

1. Виділяємо термінальні символи, які входять до неконтекстних правил та вводимо нетерміналі, які дублюють такі термінальні символи, тобто вводимо

нові нетерміналі N для кожного такого термінального символу $a \in T$.

a

2. Далі всі неконтекстні правила з P замінюємо на нові правила, в котрих замість термінальних символів стоять їх нетермінальні дублери. Додавляємо нові правила $Na \rightarrow a$. Очевидно, що отримана граmatic G' еквівалентна початковій.

3. Побудуємо граmaticу G'' , яка буде контекстною та еквівалентною G' .

Спочатку пронумеруємо усі неконтекстні правила граматики G' (контекстні не змінюємо). Розглянемо неконтекстне правило G' з номером k виду $\alpha \rightarrow \beta$, де $\alpha =$

$A_1 A_2 \dots A_n$, $n > 1$. Це правило замінимо на наступну множину нових

правил:

1 2 n-1 n

$A_1 A_2 \dots A_{n-1} A_n \rightarrow N_k A_1 A_2 \dots A_{n-1} A_n$

$N_k A_1 A_2 \dots A_{n-1} A_n \rightarrow N_k A_1 A_2 \dots A_{n-1} A_n$

$N_k A_1 A_2 \dots A_{n-1} A_n \rightarrow N_k A_1 A_2 \dots A_{n-1} A_n$

$$\begin{array}{ccccccc}
N & A_3 & \dots & A & N & \rightarrow & N & A_4 & \dots & A & N \\
kl & & & & n-1 & & kr & kl & & & n-1 & kr \\
\\
\dots & & & & \dots & & & \dots & & & & \\
N & A & N & \rightarrow & N & N & & & & & & \\
kl & n-1 & kr & & kl & kr & & & & & & \\
\\
N & N & \rightarrow & N & N & & & & & & & \\
kl & kr & & kp & kr & & & & & & & \\
\\
N & N & \rightarrow & N & \beta & & & & & & & \\
kp & kr & & kp & & & & & & & &
\end{array}$$

$N_{kr} \rightarrow \epsilon$.

(Коментар. Ідея побудови наведених правил полягає у наступному: спочатку виставляємо лівий N_{kl} та правий маркери N_{kr} вибраного правила, далі

видаляємо всі інші символи лівої частини продукції, потім переходимо до породження правої частини, але перед цим вносимо про це інформацію за допомогою N_{kr} . Наприкінці породжуємо в контексті N_{kr} праву частину продукції, а сам символ N_{kr} видаляємо.)

Можна довести, що вказана послідовність правил еквівалентна одному початковому правилу. Дійсно, ця послідовність індукує (майже детерміновано) однозначний вивід. Тільки правило $N_{kr} \rightarrow \epsilon$ порушує однозначність. Але якщо застосувати до правила $N_{kr} N_{kr} \rightarrow N_{kr} \beta$, то не буде можливості позбавитися нетермінала N_{kr} . Тому справедлива наступна теорема.

Теорема 4.2. Для довільної породжуючої граматики G_0 існує граMATика G_C у загально-контекстній формі, яка породжує ту саму мову, тобто $L(G_0) = L(G_C)$.

0 C

Визначення 4.22. Довільні граматики G_1 та G_2 називаються еквівалентними,

1 2

якщо вони породжують одну й ту саму мову, тобто $G_1 \approx G_2 \Leftrightarrow L(G_1) = L(G_2)$.

1 2

Цілком очевидно, що так введене відношення \approx є відношенням еквівалентності (рефлексивним, симетричним та транзитивним відношенням).

n n n

Приклад 4.13. Побудуємо за граMATикою G_3 , що породжує мову $L(G_3) = \{a^n b^n c^n \mid n \geq 0\}$ еквівалентну їй загально-контекстну граMATику G_3 .

C

ГраMATика G_3 має наступні правила:

- P1: $S \rightarrow aBSc$
- P2: $S \rightarrow \epsilon$
- P3: $Ba \rightarrow aB$

P4: $Bb \rightarrow bB$

P5: $Bc \rightarrow bc$

З них неконтекстними є правила P3 та P4. Спочатку перетворимо правило $Ba \rightarrow aB$ (правило P3) в послідовність контекстних правил. Отримуємо:

$Ba \rightarrow aB$ замінюємо на $BA \rightarrow A B, A \rightarrow a$.

a a a

Далі замість $BA \rightarrow A B$ вводимо послідовність правил

a a

$BA \rightarrow N A$
a 3l a

$N A \rightarrow N N$
3l a 3l 3r

$N3lN3r \rightarrow N3pN3r$
 $N N \rightarrow N A B$
3p 3r 3p a

$N3p \rightarrow \epsilon$

Аналогічно вчинимо з правилом P4: $Bb \rightarrow bB$. Після цих перетворень отримуємо нову граматику G3, в якій пронумеруємо нові правила:

C

P1: $S \rightarrow AaBSc,$

P2: $S \rightarrow \epsilon$

P31: $A \rightarrow a$

a

P32: $BA \rightarrow N A$
a 3l a

P33: $N A \rightarrow N N$
3l a 3l 3r

P34: $N N \rightarrow N N$
3l 3r 3p 3r

P35: $N N \rightarrow N A B$
3p 3r 3p a

P36: $N3p \rightarrow \epsilon$

P41: $B \rightarrow b$

b

P42: $BB \rightarrow N B$
b 4l b

P43: $N B \rightarrow N N$
4l b 4l 4r

P44: $N N \rightarrow N N$
4l 4r 4p 4r

P45: $N N \rightarrow N B B$
4p 4r 4p b

P46: $N4p \rightarrow \epsilon$

P5: $Bc \rightarrow Bbc$

Побудуємо декілька виводів у цій граматиці, тобто, говорячи в термінах програмування, зробимо «тестування» побудованої граматики.

P 41 P 2 P 1 P 2 P 5 P 31

Очевидно, що $S \Rightarrow \varepsilon$ та $S \Rightarrow A BSc \Rightarrow A Bc \Rightarrow A Bc \Rightarrow aBc \Rightarrow abc$. Побудуємо вивід

ланцюжка $a b c$. Маємо,

P 33 P 1 P 1 P 2 P 5 P 32

$S \Rightarrow A BSc \Rightarrow A BA BSc \Rightarrow A BA Bcc \Rightarrow A BA Bcc \Rightarrow A N A Bcc \Rightarrow$
 $a a a a a a a a a 31 a$

P 42 P 34 P 35 P 36

$A N N Bcc \Rightarrow A N N Bcc \Rightarrow A N A BBcc \Rightarrow A A BBcc \Rightarrow$
 $a 31 3r b a 3p 3r b a 3p a b a a b$
P 43 P 44 P 45 P 46

$A A N Bcc \Rightarrow A A N N cc \Rightarrow A A N N cc \Rightarrow A A N B Bcc \Rightarrow A$
 $A B Bcc \Rightarrow$
 $a a 4l b a a 4l 4r a a 4p 4r a a 4p b$
 $a a b$
P 41 P 41 P 31 P 31

$A A B B cc \Rightarrow A A B bcc \Rightarrow A A bbcc \Rightarrow A abbcc \Rightarrow aabbcc$.
 $a a b b a a b a a a$

Еквівалентність нескорочуючих та контекстно-залежних граматики доводиться аналогічним чином.

Це дозволяє нам говорити про те, що різні варіанти визначень граматики (породжуючі та контекстні) є еквівалентними, і що ієрархія Хомського досить стабільна відносно варіантів граматики.

Перейдемо до дослідження зв'язків класів породжуючих граматики з класами формалізмів розпізнання (сприйняття), в першу чергу, з класами автоматів різного типу.

4.7. Автоматні формалізми сприйняття мов

Дуальним методом до породження мов є їх сприйняття (розпізнавання). Цей метод

також може уточнюватися на основі поняття транзиційної системи.

Відмінність

полягає у тому, що початковий стан містить ланцюжок, для якого

потрібно перевірити його належність певній мові, а заключний стан говорить про належність (або неналежність) мові.

Існують різні уточнення методів сприйняття. Одним з них є уточнення, яке

можна отримати з породжуючих граматики простим оберненням застосування правил, тобто замість правила породжуючої граматики $\alpha \rightarrow \beta$ розглядати правило

сприймаючої граматики $\beta \rightarrow \alpha$. Вивід у сприймаючій граматиці тоді треба вести від термінального

ланцюжка до аксіоми. Такий метод уточнення нічого суттєво нового не привносить, хоча він часто розглядається в теорії формальних граматики як метод висхідного аналізу. Тому важливо визначити інші формалізми сприйняття мов. З цієї метою часто обираються формалізми автоматів (машин) різного типу.

Автомати характеризуються тим, що відбувається розподіл на інформацію, що зберігається у пам'яті певного типу, та на інформацію, що керує процесом функціонування автомату і яка задається керуючим пристроєм автомату. Такий розподіл корисний тим, що робить функціонування автомату більш детермінованим. Інакше кажучи, формалізм автоматів більш пристосований до керування виводом ніж формалізм граматики, де керування виводом дуже слабке.

Автомат у його загальній формі має пам'ять (як правило, це потенційно нескінченна стрічка), має пристрій керування (задається скінченною множиною станів), та керуючу голівку, яка «працює» з певним елементом пам'яті. Головним для автомату є спосіб його функціонування, який задається переходами із стану в стан, зміною пам'яті та рухом голівки. Ці переходи залежать від поточного стану керування та змісту елемента пам'яті, на який «дивиться» голівка.

Є різні варіанти визначень автоматів. Розглянемо ті з них, що відповідають класифікації мов за Хомським.

Найбільш потужними за сприймаючою силою є автомати, що називаються машинами Тьюрінга.

4.7.1. Машини Тьюрінга

Машина Тьюрінга M складається з таких частин.

1. Керуючий пристрій, який може приймати певний стан із скінченної множини Q .
2. Стрічка (потенційно) нескінченної довжини, розділена на комірки, в яких розміщена вхідна інформація у вигляді символів деякого алфавіту A (як правило, $QA = \emptyset$). В кожній комірці записано по одному символу з алфавіту.

Виділяється особливий символ $\# \in A$, який інтерпретуємо як "порожній символ", причому в кожен даний момент стрічка містить лише скінчену кількість символів, відмінних від $\#$. Вважаємо також, що всі такі "непорожні" символи записані підряд.

3. Голівка читання-запису забезпечує обмін інформацією між стрічкою і керуючим пристроєм. У кожний момент часу голівка може працювати тільки з однією коміркою стрічки. Голівка може

- замінити прочитаний символ іншим символом алфавіту A ,
- переміщуватись на одну позицію вправо чи вліво, чи
- залишатися на місці.

Робота машини задається спеціальними правилами переходу (командами), що називаються програмою машини. Програма складається з команд виду $qa \rightarrow pbR$, $qa \rightarrow pbL$, $qa \rightarrow pb$ ($q, p \in Q$, $a, b \in A$, L, R - додаткові символи). Інтерпретація команд наступна:

- Виконання команди $qa \rightarrow pbR$. Якщо керуючий пристрій знаходиться у стані q , а голівка оглядає комірку на стрічці, у якій записано символ a , то керуючий пристрій переходить у стан p , голівка у комірку записує символ b і сама зміщується на одну комірку вправо.

- Виконання команди $qa \rightarrow pbL$. Відрізняється від попереднього лише тим, що голівка зміщується вліво.
- Виконання команди $qa \rightarrow pb$. Відрізняється від попередньої команди тим, що голівка залишається на місці.

У подальшому не будемо акцентувати увагу на керуючому пристрої та голівці і говорити просто, що машина Тьюрінга змінює свій стан та рухається вліво або вправо.

У фіксований момент часу поточна інформація задається за допомогою конфігурації машини Тьюрінга. Конфігурацією машини Тьюрінга зазвичай називається трійка $(q, \#w, v\#)$, де $q \in Q$, $w, v \in A^*$. Конфігурація описує повний стан машини й інтерпретується таким чином:

- q - стан керуючого пристрою,
- $\#w$ - ланцюжок, записаний на стрічні машини вліво від голівки,
- $v\#$ - ланцюжок, записаний на стрічні машини вправо від голівки,

включаючи символ, який оглядає голівка.

Зважаючи на те, що будемо перетворювати команди машини Тьюрінга у правила граматики, конфігурації в подальшому будемо задавати у вигляді ланцюжка $\#wqv\#$ (тому тут важливо, щоб $Q \cap A = \emptyset$).

Оскільки машина Тьюрінга є конкретизацією транзиційної системи, то у визначення вводиться початковий стан $q \in Q$ та множина заключних станів $F \subseteq Q$.

0

Звичайним чином вводиться відношення безпосереднього виводу \Rightarrow (часто

позначається \vdash) та рефлексивне транзитивне замикання цього відношення \Rightarrow^* (\vdash^*).

Введені поняття дають можливість формального визначення машин Тьюрінга.

Визначення 4.23. Машиною Тьюрінга M називається послідовність параметрів $(Q, A, \#, \delta, q_0, F)$, де

- Q - скінченна множина станів,
- A - скінченний алфавіт ($Q \cap A = \emptyset$),
- $\#$ - символ з A («порожній» символ),
- δ - скінченна множина команд $qa \rightarrow pbR$, $qa \rightarrow pbL$, $qa \rightarrow pb$ ($q, p \in Q$, $a, b \in A$,

L, R

- додаткові символи),

- $q_0 \in Q$ - початковий стан із Q ,
- F - підмножина фінальних станів із Q .

Зауважимо, що є багато варіантів визначення машин Тьюрінга, наприклад, може бути декілька стрічок, порожній символ можна явно не вводити в алфавіт і вважати

його однаковим для класу машин Тьюрінга, не вводити заключних станів і т. ін. Всі

такі визначення, як правило, є еквівалентними, з точки зору сприйняття мов.

Наведене визначення задає екстенціонал поняття машини Тьюрінга, тобто клас одиничних машин. Що стосується інтенціоналу, то він обумовлений призначенням

машин, орієнтованих на сприйняття мов, яке засновано на відношенні \vdash (\Rightarrow)

T T

безпосереднього переходу від однієї конфігурації до іншої.

Визначення 4.24. Конфігурацією машини Тьюрінга $M = (Q, A, \#, \delta, q_0, F)$

називається ланцюжок виду $\#wqv\#, q \in Q, w, v \in A^*$.

Визначення 4.25. Нехай $M = (Q, A, \#, \delta, q, F)$ - машина Тьюрінга. Конфігурація

0

$\#wqv\#$ безпосередньо переходить

- у конфігурацію $\#wcbpv\#,$ якщо виконується команда $qa \rightarrow pbR,$
- у конфігурацію $\#wrcbv\#,$ якщо виконується команда $qa \rightarrow pbL,$ та
- у конфігурацію $\#wcpbv\#,$ якщо виконується команда $qa \rightarrow pb$

($a, b, c \in A, w, v$

$\in A^*, q, p \in Q$).

Ми будемо відношення безпосереднього переходу також називати відношенням виводу конфігурацій, щоб продемонструвати його єдність із відношенням виводу у породжувачих граматиках та формальних системах. Послідовність конфігурацій, пов'язаних відношенням безпосереднього виводу, називається протоколом машини Тьюрінга.

Визначення 4.26. Рефлексивне транзитивне замикання відношення \vdash позначаємо \vdash^* .

Початкові конфігурації мають вид $\#q v\#,$ фінальні - $\#w q w \#, q \in F.$

0

1 F 2

F

I, нарешті, головне смислове визначення.

Визначення 4.27. Мова, яка допускається машиною Тьюрінга $M = (Q, A, \#, \delta, q_0, F),$ є множина ланцюжків

$$L(M) = \{ w \mid \#q w \# \vdash^* \#w q w \#, q \in F, w, w, w \in A^* \}$$

T

0

T

1

F

2

F

1

2

Таким чином, поняття машини Тьюрінга є єдністю двох аспектів: екстенціонального та інтенціонального. Екстенціональний аспект задається як клас машин Тьюрінга, визначених відповідними шістьками параметрів, інтенціональний аспект задається уніформним визначенням відношення безпосередньої вивідності, та мови, що сприймається (допускається, розпізнається) машиною.

~

Приклад 4.14. Побудуємо машину Тьюрінга, яка допускає мову $\{ r^i \mid r \in \{a, b\}^* \}.$

Ідея програми наступна:

1. У початковому стані q_0 машина читає символ $a, b,$ або $\#.$ У перших двох випадках машина стирає прочитані символи та «запам'ятовує» прочитаний

символ відповідно у стані q або у стані q . Останній випадок означає, що на

a b

вхідній стрічці - порожній ланцюжок і тому машина переходить у заключний стан q .

F

2. Далі голівка рухається у правий бік до кінця ланцюжка.

Прочитавши

порожній символ #, голівка повертається на одну комірку (клітинку) вліво та

переходить у стан q зі стану q , або у стан q зі стану q .

Тут індекси $-a$ та $-b$

$-a$ a $-b$

b

означають, що машині потрібно стерти відповідно символ a або b .

3. Якщо правий символ ланцюжка співпадає з символом, «запам'ятованим» як

такий, що потрібно стерти, то він стирається, машина переходить у стан

q ;

L

якщо ні, то робота машини блокується (немає застосовних правил) і вхідний

ланцюжок не належить нашій мові.

4. У стані qL голівка рухається вліво, поки не дійде до символу #. Далі голівка

переходить у стан $q0$ та зміщується у правий бік. Один цикл перевірки

завершено. Робота продовжується наступним

циклом перевірки або

завершується, якщо все перевірено.

Наведений алгоритм дозволяє побудувати машину Тьюрінга

$M = (\{ q, q, q, q, q, q, q, q \}, \{ a, b \}, \#, \delta, q, \{ q \})$,

$0 \quad a \quad -a \quad b \quad -b \quad L \quad F \quad 0$

F

де δ - множина наступних команд:

1. $q \ a \rightarrow q \ \#R$
 $0 \quad a$
2. $q \ b \rightarrow q \ \#R$
 $0 \quad b$
3. $q \ a \rightarrow q \ aR$
 $a \quad a$
4. $q \ b \rightarrow q \ bR$
 $a \quad a$
5. $q \ \# \rightarrow q \ \#L$
 $a \quad -a$
6. $q \ a \rightarrow q \ \#L$
 $-a \quad L$
7. $q \ a \rightarrow q \ aR$
 $b \quad b$
8. $q \ b \rightarrow q \ bR$

- b b
9. q #→q #L
 b -b
10. q b→q #L
 -b L
11. q a→q aL
 L L
12. q b→q bL
 L L
13. q #→q #R
 L 0
14. q #→q #
 0 F

Розглянемо на прикладі сприйняття слів цієї машиною Тьюрінга. Візьмемо слово

#abba#. Побудуємо протокол його сприйняття:

#q abba#⇒#q bba#⇒#bq ba#⇒#bbq a#⇒#bbaq #⇒
0 a a a a

#bbq a#⇒#bbq #⇒ # bq b#⇒# q bb#⇒ q #bb#⇒
-a L L L L

q bb#⇒# q b#⇒# bq #⇒ #q b#⇒#q #⇒#q #⇒#q #
0 b b -b L 0 F

4.7.2. Еквівалентність машин Тьюрінга та породжувачих граматик

Наведений у попередньому прикладі протокол машини Тьюрінга підказує ідею побудови породжувачої граматики за програмою машини. Таку побудову зробимо в два етапи: спочатку за програмою побудуємо продукції переходу (перетворення) конфігурацій машини Тьюрінга, потім зробимо обернення побудованих продукцій та додамо правило для аксіоми та правила породження і видалення порожніх символів.

Отримана породжувача граMATика буде еквівалентна вибраній машині Тьюрінга.

Кожна команда машини Тьюрінга породжує наступні правила перетворення конфігурацій:

- Команда виду $qa \rightarrow pbR$ породжує правило $qa \rightarrow bp$.
- Команда виду $qa \rightarrow pbL$ породжує множину правил перетворення $\{ sqa \rightarrow pcb \mid c \in A \}$.
- Команда виду $qa \rightarrow pb$ породжує правило $qa \rightarrow pb$.

На другому етапі побудови обернемо отримані правила (перетворення виду $\alpha \rightarrow \beta$ замінимо на правило породжувачої граматики $\beta \rightarrow \alpha$), додамо правило для аксіоми

$S \rightarrow \#q \#$. Враховуючи, що за таких побудованих простих правилах можуть бути зайві

$\#$, переведемо їх в пустий ланцюжок ε правилом $\# \rightarrow \varepsilon$, або створимо у разі необхідності додаткові порожні символи $\#$ правилом $\# \rightarrow \#\#$. Нарешті, потрібно буде видалити початковий стан, щоб отримати (породити) початковий ланцюжок правилом $q \rightarrow \varepsilon$.

Таким чином, буде створена граматики $G = (N, T, P, S)$, яка має наступні

$G = (N, T, P, M)$

параметри:

- $N = \{S\} \cup Q \cup \{\#\}$ ($S \notin Q \cup \{\#\}$).
- $T = A \setminus \{\#\}$.
- $S \in N$.
- P будується за вказаним вище алгоритмом.

Продемонструємо наведений алгоритм прикладом.

Приклад 4.15. Візьмемо машину Тьюрінга, запропоновану в попередньому прикладі, та побудуємо еквівалентну їй граматику. Отримаємо граматику $G = (\{S, Q, M, \#\}, \{a, b\}, P, S)$.

$M = 0$

$Q = \{q, q', q'', q''', \#\}, \{a, b\}, P, S)$.
 $a \quad -a \quad b \quad -b \quad L \quad F$

Потрібні перетворення команд наведено в таблиці, третій стовпчик якої задає продукції P .

Таблиця 4.1

Команди машини Продукції породжуючої Тьюрінга граматики	Правила перетворення конфігурацій
0. $S \rightarrow \#q \#$ F	
0. $\# \rightarrow \#\#$ $\#$	
1. $q \underset{0}{a} \rightarrow q \underset{a}{\#R}$ 1. $\#q \rightarrow \#q a$ $a \quad 0$	1. $\#q \underset{0}{a} \rightarrow \#q \underset{a}{\#}$
2. $q \underset{0}{b} \rightarrow q \underset{b}{\#R}$ 2. $\#q \rightarrow \#q b$ $b \quad 0$	2. $\#q \underset{0}{b} \rightarrow \#q \underset{b}{\#}$
3. $q \underset{a}{a} \rightarrow q \underset{a}{aR}$ 3. $aq \rightarrow q a$ $a \quad a$	3. $q \underset{a}{a} \rightarrow aq \underset{a}{\#}$

4. $q \xrightarrow{a} b \xrightarrow{a} bR$
 4. $bq \rightarrow q b$
 a a

5. $q \xrightarrow{a} \# \rightarrow q \quad \#L$
 5. $q a\# \rightarrow aq \#$
 -a a

6. $q \xrightarrow{-a} a \rightarrow q \quad \#L$
 6. $q a\# \rightarrow aq \quad a; q b\# \rightarrow$
 L -a L

$bq \quad a; \quad q \## \rightarrow \#q \quad a$
 -a L -a

7. $aq \rightarrow q a$

b b
 7. $q \xrightarrow{b} a \rightarrow q \quad aR$
 b b

8. $bq \rightarrow q b$

b b
 8. $q \xrightarrow{b} b \rightarrow q \quad bR$
 b b

9. $q b\# \rightarrow bq \#$

-b b
 9. $q \xrightarrow{b} \# \rightarrow q \quad \#L$
 b -b

10. $q a\# \rightarrow aq \quad b; q b\# \rightarrow$
 L -b L

10. $q \xrightarrow{-b} b \rightarrow q \quad \#L$
 -b L
 $bq \quad b; q \## \rightarrow \#q \quad b$

-b L -b

11. $q aa \rightarrow aq a; q ba \rightarrow$

L L L

$bq \quad a; q \#a \rightarrow \#q \quad a;$

L L L

4. $q \xrightarrow{a} b \rightarrow bq$
 a a

5. $aq \xrightarrow{a} \# \rightarrow q \quad a\#$
 a -a

6. $aq \quad a \rightarrow q \quad a\#; bq$
 -a L -

$a \rightarrow q \quad b\#;$
 a L $\#q$ -

$a \rightarrow q \quad \##;$
 a L

7. $q \xrightarrow{b} a \rightarrow aq$
 b b

8. $q \xrightarrow{b} b \rightarrow bq$
 b b

9. $bq \xrightarrow{b} \# \rightarrow q \quad b\#$
 b -b

10. $aq \quad b \rightarrow q \quad a\#; bq$
 -b L -

$b \rightarrow q \quad b\#;$
 b L

$\#q \quad b \rightarrow q \quad \##;$
 -b L

11.q a→q aL
 12.q ab → aq b; q bb →
 L L L

bq b; q #b → #q b;
 L L L

13.#q → q #
 0 L

12.q b→q bL
 L L
 14.#q #→#q #
 F 0

15.q →ε
 0

13.q #→q #R
 L 0
 16. #→ε

14.q #→q #
 0 F

11.aq a→q aa;
 L L

bq a→q ba;
 L L

#q a→q #a
 L L

12.aq b→q ab;
 L L

bq b→q bb;
 L L

#q b→q #b
 L L

13.q #→#q
 L 0

14.#q #→#q #
 0 F

Правила, отримані перетворенням команди з номером n (n=6, 10, 11, 12), будемо далі нумерувати як n(1), n(2), n(3).

Продемонструємо коректність правил виводу для породження ланцюжка abba (індекси вказують на застосоване правило):

S ⇒ #q # ⇒ #q # ⇒ #q # ⇒ #q ## ⇒ ##q b# ⇒ #q b# ⇒ #q b# ⇒
 #bq # ⇒
 0 F 14 0 13 L 0# L 10(3) -b 16 -b 9
 b 8

#q b# ⇒ ##q b# ⇒ #q #bb# ⇒ ##q bb# ⇒ #q bb# ⇒ #
 bq b# ⇒
 b 0# 0 13 L 12(3) L 16 L 12(2)
 L 0#

	#bq b## \Rightarrow	# bbq	a## \Rightarrow	#bbaq # \Rightarrow	#bbq a## \Rightarrow	#bq ba## \Rightarrow	#q bba## \Rightarrow	
a	L	6(2)	-a	5	a	3	a	4
	1						a	4

#q abba## \Rightarrow	q abba## \Rightarrow	q abba.
0	16	0
		16
		0

Зауважимо, що в процесі виводу застосовувались правила породження та знищення порожнього символу #.

Наведені вище міркування та приклад дозволяють сформулювати наступне твердження.

Теорема 4.3. За кожною машиною Тьюрінга M можна побудувати еквівалентну їй породжуючу граматику G, що породжує ту ж мову, яку сприймає M, тобто:
 $L(M) = L(G)$.

T

Ця теорема може бути обернена.

Теорема 4.4. За кожною породжуючою граматикою G можна побудувати еквівалентну їй машину Тьюрінга M, що сприймає ту ж мову, яку породжує G, тобто:
 $L(M) = L(G)$.

Доведення може бути конструктивним, коли машиною Тьюрінга моделюється зворотній до породження процес сприйняття ланцюжка, або можна скористатися

тезою Чорча, що стверджує існування машини Тьюрінга для формального подання інтуїтивних алгоритмів. Тут деталі такої побудови описувати не будемо.

З наведених теорем випливає, що граматики типу 0 породжують усі рекурсивно-зліченні множини (мови) в алфавіті A. Тим самим клас породжуючих граматик має таку ж виразну потужність, як і інші універсальні моделі алгоритмів.

Відзначимо, що є різні варіанти машин Тьюрінга, наприклад, машини з різною кількістю стрічок, з різними обмеженнями на команди. Зокрема, виділяють детерміновані (не існує команд з однаковими лівими частинами та різними правими частинами), та недетерміновані машини Тьюрінга (існують команди з однаковими лівими частинами та різними правими частинами). Клас формальних мов, що сприймаються машинами Тьюрінга, є однаковий для детермінованих та недетермінованих машин, та для машин з різною кількістю стрічок.

4.7.3. Лінійно-обмежені автомати

Лінійно-обмежені автомати можуть розглядатися як обмежені машини Тьюрінга, для яких довжина ланцюжка на стрічці завжди лінійно обмежена довжиною вхідного ланцюжка.

Визначення 4.28 Машина Тьюрінга $M = (Q, A, \#, \delta, q, F)$ називається лінійно-

0

обмеженим автоматом, якщо існує число k, що для будь якого ланцюжка $v \in A^*$

довжини n, з умови $\#q v \# \xrightarrow{*} \#w q w \#$ ($q \in Q, w \in A^*$) випливає, що $|w| \leq k \cdot n$.

0 1 2 1, 2

1 2

Наведену умову важко перевірити, маючи машину Тьюрінга, тому наведене визначення часто спрощують, вимагаючи, щоб $k=1$, тобто, щоб голівка машини

Тьюрінга не могла записувати нові символи за межами вхідного ланцюжка. Є й інші варіанти визначення лінійно-обмежених автоматів.

Використовуючи методи, наведені в попередньому підрозділі, можемо за лінійно-обмеженим автоматом побудувати еквівалентну породжуючу граматику типу 1, і

навпаки, за граматику типу 1 побудувати лінійно-обмежений автомат.

Таким

чином, справедливе наступне твердження.

Теорема 4.5. За кожним лінійно-обмеженим автоматом можна побудувати

еквівалентну йому породжуючу граматику типу 1, і

навпаки, за кожною

породжуючою граматику типу 1 можна побудувати еквівалентний їй

лінійно-

обмежений автомат.

Говорячи більш просто, теорема стверджує, що клас лінійно-обмежених автоматів еквівалентний класу породжуючих граматик типу 1.

4.7.4. Магазинні автомати

Магазинні автомати (автомати з магазинною пам'яттю, МП-автомати, стекові

автомати) широко використовуються в програмуванні, бо дозволяють моделювати

різні важливі алгоритми, які пов'язані з компіляцією та

інтерпретацією мов

програмування, обробки складних структур даних - дерев, списків і т.

ін. Такі

автомати можна розглядати як обмежені машини Тьюрінга, що мають вхідну

стрічку, на якій голівка може тільки читати символи і рухатися тільки в один

бік, та

робочу стрічку, яку відповідна голівка може обробляти лише з однієї сторони. Як

і в

попередніх випадках є різні варіанти визначень магазинних автоматів.

Наведемо

тут просте екстенціональне визначення, яке разом з тим є еквівалентним іншим

визначенням за класом мов, що розпізнаються.

Визначення 4.29. Магазинний автомат - це шістка $M = (Q, A, \Gamma, \delta, q_0, F)$, де

- Q - скінченна множина станів,

- A - скінченний вхідний алфавіт ($Q \cap A = \emptyset$),

- Γ - скінченний магазинний алфавіт,

- δ - скінченне відношення переходів (скінченне

відображення

$\delta: Q \times A \times \Gamma \rightarrow Q \times \Gamma^*$),

- q_0 - початковий стан із Q ,

- F - підмножина фінальних (заключних) станів із Q .

Виконання команди виду $(q, a, Z) \rightarrow (p, \gamma)$ означає, що знаходячись у стані q , та

оглядаючи символ a на вхідній стрічці і символ Z у магазині, автомат

переходить у

стан p , стирає символ a та пересуває голівку до наступного символу

на вхідній

стрічці, а замість символу Z записує у магазин ланцюжок γ . Зауважимо, що в

інших

модифікаціях магазинних автоматів перехід у новий стан може відбуватися

без

стирання символу із вхідної стрічки (так званий ϵ -перехід).

Конфігурацію магазинного автомату можна задати як трійку (q, w, ζ) , де $q \in Q$, $w \in$

A^* , $\zeta \in \Gamma^*$). Відношення безпосереднього переходу $|-$ задається так: $(q, aw', Z\zeta') |-(p, w', \gamma\zeta')$, якщо команда $(q, a, Z) \rightarrow (p, \gamma)$ належить δ .

Аналогічно тому, як це робилося раніше, визначаються й інші необхідні поняття, пов'язані з магазинним автоматом: протоколи та мова, що сприймається таким автоматом.

Для нас важливим є наступне твердження.

Теорема 4.6. За кожним магазинним автоматом можна побудувати еквівалентну йому породжуючу граматику типу 2 (контекстно-вільну граматику), і навпаки, за кожною породжуючою граматикою типу 2 можна побудувати еквівалентний їй магазинний автомат.

Зважаючи на те, що синтаксис мов програмування, як правило, задається граматиками типу 2, теорема стверджує, що алгоритми обробки програм можуть базуватися на магазинних автоматах.

4.7.5. Скінченні автомати

Скінченні автомати можна розглядати як обмежені машини Тьюрінга, що мають одну вхідну стрічку – голівка може тільки читати символи і рухатися тільки в один бік. Тому такі автомати не мають необмеженої пам'яті. Екстенціональне визначення наступне.

Визначення 4.30. Скінченний автомат – це п'ятірка $M = (Q, A, \delta, q_0, F)$, де

0

- Q – скінченна множина станів,
- A – скінченний вхідний алфавіт ($Q \cap A = \emptyset$),
- δ – скінченне відношення переходів (скінченне відображення $\delta: Q \times A \rightarrow Q$),
- $q_0 \in Q$ – початковий стан із Q ,
- F – підмножина фінальних (заклучних) станів із Q .

Всі основні загальні (інтенціональні) поняття визначаються подібно до того, як це було зроблено для машин Тьюрінга та магазинних автоматів, тому тут їх наводити не будемо.

Основним є наступний результат.

Теорема 4.7. За кожним скінченним автоматом можна побудувати еквівалентну йому породжуючу граматику типу 3 (ліволінійну чи праволінійну граматику), і навпаки, за кожною породжуючою граматикою типу 3 можна побудувати еквівалентний їй скінченний автомат.

Відзначимо, що детермінованість чи недетермінованість скінченного автомату не впливає на клас мов, що сприймаються автоматами.

Теорія скінченних автоматів розроблена досить детально, для них побудовані алгоритми еквівалентних перетворень, мінімізації та ін. Тут наведемо лише один результат, що стосується алгебраїчного подання скінченно-автоматних мов. Такі мови ще називаються регулярними мовами.

Регулярні мови задаються виразами (термами) регулярної алгебри мов, що має операції об'єднання, конкатенації та ітерації.

Регулярні вирази можна визначити індуктивно.

Базис складається з трьох визначень.

1. Константи ε та \emptyset є регулярними виразами.
2. Якщо a – довільний символ алфавіту, то a – регулярний вираз.
Зауважимо, що часто в написанні розрізняють символ алфавіту та відповідний вираз. Тут це не робимо, щоб не ускладнювати текст.

3. Якщо X – змінна, то X – регулярний вираз.

Крок індуктивної побудови. Індуктивний крок складається з чотирьох визначень, по одному для трьох операторів та для введення дужок.

1. Якщо E та F – регулярні вирази, то $E+F$ – регулярний вираз.
2. Якщо E та F – регулярні вирази, то EF – регулярний вираз.
Зауважимо, що для позначення оператора конкатенації – як операції над мовами, так і оператора в регулярному виразі – можна використовувати крапку.
3. Якщо E – регулярний вираз, то E^* – регулярний вираз.
4. Якщо E – регулярний вираз, то (E) – регулярний вираз.

Інтерпретація L виразів у класі мов над алфавітом A задається також індуктивно на підставі інтерпретації змінних $L : \text{Var} \rightarrow 2A^*$, де Var – множина змінних.

Інтерпретація базових виразів задається таким чином:

1. $L(\varepsilon) = \{\varepsilon\}$ і $L(\emptyset) = \emptyset$.
2. $L(a) = \{a\}$.
3. $L(X) = LV(X)$.

Інтерпретація складних виразів задається таким чином (E та F – регулярні вирази):

1. $L(E+F) = L(E) \cup L(F)$.
2. $L(EF) = L(E)L(F)$.
3. $L(E^*) = (L(E))^*$.
4. $L((E)) = L(E)$.

Ми використовуємо $L(E)$ для позначення мови, яка відповідає E . Як і в інших алгебрах, оператори регулярних виразів мають певні пріоритети, тобто оператори зв'язуються зі своїми операндами в певному порядку.

Найвищий пріоритет має оператор ітерації $*$. Далі йде оператор конкатенації. Оскільки він асоціативний, то не має значення, в якому порядку групуються послідовні конкатенації. Але, якщо необхідно групувати вирази, то робимо це, починаючи зліва. Найнижчий пріоритет має оператор об'єднання. Він також асоціативний. Для нього будемо притримуватися групування, починаючи з лівого краю виразу.

Основний результат стосовно регулярних мов є наступний.

Теорема 4.8. За кожним скінченим автоматом можна побудувати еквівалентний йому регулярний вираз, і навпаки, за кожним регулярним виразом можна побудувати еквівалентний йому скінченний автомат.

Наведений результат дозволяє використовувати різні формалізми (граматики типу 3, скінченні автомати, регулярні вирази) для дослідження властивостей регулярних мов. Такі мови мають гарні властивості замкненості (відносно об'єднань, перетину,

доповнень, конкатенації, ітерації та деяких інших операцій). Ще одну важливу групу властивостей регулярних мов становлять «властивості розв'язності». За їх допомогою можна з'ясувати, чи визначають два різні автомати одну і ту саму мову. Розв'язність цієї задачі дозволяє мінімізувати автомати, тобто за даним автоматом знайти еквівалентний йому з найменшою можливою кількістю станів. Задача мінімізації має велике значення при проектуванні інтегральних схем.

4.8. Методи подання синтаксису мов програмування

Граматики типу 2 (контекстно-вільні граматики) відіграють велику роль у формалізації мов програмування. Справа в тому, що вони задають чітке подання деякого синтаксичного поняття як структури, що складається з певних частин. Ці частини, у свою чергу, мають частини і так далі. Тобто контекстно-вільні граматики подають ієрархічну організацію синтаксичного поняття. Це відповідає і композиційній структурі програм. Саме тому вивченню цього класу граматик буде приділено більшу увагу. Щоб продемонструвати зв'язок з мовами програмування більш чітко, розглянемо методи подання синтаксису мов програмування. Найбільш відомим методом є нормальні форми Бекуса-Наура (БНФ). Цей формалізм (метамова) широко використовується як під час подання синтаксису мов програмування, так і під час вивченні природних мов.

4.8.1. Нормальні форми Бекуса-Наура

Ці форми були запропоновані Дж. Бекусом та П. Науром для опису синтаксису мови програмування АЛГОЛ-60. Основним призначенням форм Бекуса та Наура було подання у компактному вигляді строго формальних правил написання основних конструкцій мов програмування.

Приблизно в той самий час Ноам Хомський (1959) ввів аналогічну форму - контекстно-вільну граматику - для визначення синтаксису природної мови.

БНФ складається зі скінченної множини правил, які визначають формальну мову (зокрема, синтаксис мов програмування).

Одним із класів об'єктів, що використовуються в БНФ, є символи описуваної мови. Другим класом об'єктів БНФ виступають мета змінні (нетермінальні символи), значеннями яких є ланцюжки основних символів. Для опису синтаксису мов програмування використовують велику кількість нетермінальних символів, тому для наочності їх подають як комбінації слів природної мови, що поміщені в кутові дужки. Ліву частину правил відділяють від правої частини знаком ::=, крім того, правила з однаковою лівою частиною записуються як одне правило, при цьому праві частини (альтернативи) розділяються вертикальною рискою |.

Приклад 4.16. Синтаксис операторів мови SIPL задається наступною БНФ:

```
<оператор > ::= <змінна >:=<вираз > |  
                <оператор > ; <оператор > |  
                if <умова > then <оператор > else <оператор > |
```

```

while <умова > do <оператор > |
begin <оператор > end |
skip

```

За кожною БНФ легко побудувати контекстно-вільну граматику, співставляючи правило БНФ виду $A ::= \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ сукупність правил граматики $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \dots, A \rightarrow \alpha_n$.

4.8.2. Модифіковані нормальні форми Бекуса-Наура

Для отримання більш наочних та компактних описів синтаксису застосовують модифіковані БНФ. Найчастіше передбачається введення спеціальних позначень для ітерації та альтернативи.

Наприклад, список якихось елементів задається БНФ

$$\langle \text{список} \rangle ::= \langle \text{елемент} \rangle \langle \text{список} \rangle \mid \varepsilon \varepsilon$$

У модифікованій БНФ можна записати

$$\langle \text{список} \rangle ::= \{ \langle \text{елемент} \rangle \}^*$$

Така форма передбачає введення операції, яка називається ітерацією і позначається парю фігурних дужок із зірочкою.

Якщо певна частина синтаксичної конструкції може бути пропущена, то в модифікованих БНФ її виділяють квадратними дужками. Наприклад, замість правила БНФ

$$\langle \text{оператор} \rangle ::=$$

$$\text{if } \langle \text{умова} \rangle \text{ then } \langle \text{оператор} \rangle \text{ else } \langle \text{оператор} \rangle \mid \text{if } \langle \text{умова} \rangle \text{ then } \langle \text{оператор} \rangle$$

можна вжити наступне правило модифікованої БНФ:

$$\langle \text{оператор} \rangle ::= \text{if } \langle \text{умова} \rangle \text{ then } \langle \text{оператор} \rangle \text{ [else } \langle \text{оператор} \rangle \text{]}.$$

Зрозуміло, що для модифікованих БНФ вживати дужки « { », « } », « [» та «] », а також « * » у якості символів мови не слід, тому що вони виступають як метасимволи формалізму модифікованих БНФ. За необхідності вживання цих дужок і зірочки для них використовують спеціальні позначення.

4.8.3. Синтаксичні діаграми

Для поліпшення зорового сприйняття і полегшення розуміння синтаксичних описів, застосовують подання синтаксичних правил у вигляді синтаксичних діаграм. Такі діаграми мають одну вхідну і одну вихідну стрілки. Схематично їх можна зображувати таким чином (D – внутрішність діаграми):

D

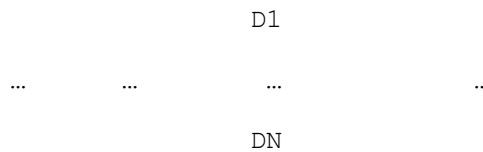
Синтаксичні діаграми є структурованими. Це означає, що вони визначаються індуктивно: є найпростіші діаграми, а більш складні будуються за допомогою певних

правил.

Найпростішими є наступні діаграми:

- Порожня діаграма
- Термінальна діаграма (a – термінальний символ)
a
- Нетермінальна діаграма (A – нетермінальний символ)
A

Далі вважаємо, що побудовані наступні діаграми:



Є три правила побудови нових діаграм з наведених:

- Послідовність
D1 DN
- Альтернатива
D1
... ..
DN
- Ітерація
D1
D2

Діаграми можуть мати назву, яка є нетермінальним символом.

Сукупність іменованих діаграм задає певну формальну мову, кожне слово якої складається з термінальних символів, які знаходяться на шляху від початкової стрілки до заключної (від вхідної до вихідної стрілки), причому нетермінальні символи замінюються на термінальні слова, які задаються діаграмою, що називається цим нетермінальним символом.

Приклад 4.17. Наведемо кілька діаграм, які задають числа:

- Цифра:
<цифра> 0

- Ціле без знака:

<ціле без знака>

<цифра>

- Дійсне число (без експоненти):

<дійсне число>

+

<ціле без знака>

<ціле

без знака>

.

-

Щоб скоротити кількість діаграм, їх часто об'єднують, замінюючи нетермінальні символи, які входять у діаграму, на побудовані для них діаграми.

За кожною БНФ можна побудувати систему синтаксичних діаграм.

Метод

побудови наступний (індукція за структурою БНФ):

- порожньому слову відповідає порожня діаграма;
- термінальному символу відповідає термінальна діаграма з цим

символом;

- нетермінальному символу відповідає нетермінальна діаграма з цим символом;

- послідовності символів, що утворюють одну з альтернатив правої частини

правила БНФ, відповідає послідовність діаграм, що задають символи цієї частини;

- альтернативам правила (тобто виразу $::= \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$)

буде відповідати

1 2

діаграма - альтернатива для діаграм, побудованих за виразами $\alpha_1, \alpha_2, \dots, \alpha_n$

, ... ,

1 2

α_n ;
n

- правилу $A ::= \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ буде відповідати деяка діаграма з іменем

A, яка

1 2 n

визначається для виразу $\alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$.

Неважко побудувати і зворотній алгоритм, який за системою синтаксичних діаграм вказаного вигляду буде еквіваленту їй БНФ.

Таким чином, справедливе наступне твердження.

Твердження 4.1. Наступні формалізми подання формальних мов: БНФ, модифіковані БНФ, контекстно-вільні граматики, синтаксичні діаграми, є еквівалентними формалізмами.

Зауважимо, що наведене твердження не сформульовано як теорема лише тому, що не було дано достатньо точного визначення формальних мов, що породжуються такими формалізмами, як БНФ, модифіковані БНФ та синтаксичні діаграми. Разом з тим надати такі формальні визначення не дуже складно (зробіть це самостійно).

Зважаючи на наведене твердження, будемо в подальшому серед указаних формалізмів розглядати переважно формалізм контекстно-вільних граматик.

4.9. Властивості контекстно-вільних граматик

У цьому підрозділі розглянемо властивості контекстно-вільних граматик (КВ-грамматик), зокрема їх еквівалентні перетворення та нормальні форми. Спочатку доведемо лему про еквівалентність граматик щодо бієктивного перейменування нетерміналів.

Лема 4.2 (про перейменування нетерміналів). Нехай задані граматика $G=(N, T, P, S)$ та бієктивне відображення $\rho: N \rightarrow N'$ множини нетерміналів N на деяку іншу

множину нетерміналів N' ($N' \cap T = \emptyset$). Тоді для граматики $G'=(N', T, S', P')$, де P' – множина правил, отриманих з P заміною нетерміналів N на відповідні нетермінали з N' , а $S'=\rho(S)$, маємо: $L(G)=L(G')$.

Для доведення леми слід скористатися тією обставиною, що кожний вивід в одній граматиці має відповідний вивід (шляхом перейменування нетерміналів) в іншій граматиці.

Зрозуміло, що умова бієктивності є суттєвою.

4.9.1. Видалення несуттєвих символів

Продовжимо розгляд очевидних, але важливих перетворень. У деяких випадках КВ-грамматика може містити символи та правила, що не вживаються для виводу термінальних ланцюжків.

Приклад 4.18. У граматиці $G=(\{S, A, B\}, \{a, b, c\}, P, S)$, де $P=\{S \rightarrow a, S \rightarrow cS, A \rightarrow b\}$, нетермінал A і термінал b не можуть з'явитися в жодному ланцюжку виведення (в жодній словоформі). Таким чином, ці символи не приймають участь у породженні ланцюжків мови $L(G)$ і правила, що їх містять, можна видалити, не змінивши мови $L(G)$.

Визначення 4.31. Нетермінал $A \in NUT$ назвемо недосяжним у граматиці $G=(N, T, P, S)$, якщо A не з'являється в жодному вивідному ланцюжку, тобто не існує виводу

виду $S \Rightarrow^* \gamma A \delta$, $\gamma, \delta \in (NUT)^*$.

Для знаходження недсяжних нетерминалів спочатку визначимо множину досяжних нетерминалів. Ця множина для заданої КВ-граматики $G=(N, T, P, S)$ легко визначається за допомогою наступних індуктивних визначень.

$$1. R_0 = \{S\}.$$

$$2. R_i = \{A \mid \exists \text{ правило } A \rightarrow \alpha B \beta \in P, \text{ що } A \in R_{i-1}, B \in N\} \cup R_{i-1} \quad (i=1, 2, \dots).$$

Оскільки множина N є скінченною, а формула для визначення R_i задає монотонне

за i відображення, то існує k ($k \geq 0$), що $R_k = R_{k+1}$. Інакше кажучи, послідовність R_0, R_1, R_2, \dots стабілізується на k -му кроці. Покладемо $R = R_k$.

Множина UR недсяжних нетерминалів задається формулою $UR = N \setminus R$.

За граматикою $G=(N, T, P, S)$ будемо граматикою $G'=(N', T', P', S')$ таким чином:

1. $N' = N \cup R$
2. $T' = T$
3. $S' = S$
4. $P' = \{A \rightarrow \alpha \in P \mid \alpha \in (R \cup T)^*\}$

Очевидною є наступна лема.

Лема 4.3. Для граматикою $G'=(N', T', P', S')$ виконуються наступні властивості:

1. $L(G') = L(G)$ (еквівалентність).
2. Для всіх $A \in N'$ існують такі ланцюжки α та β із $(N' \cup T)^*$, що

$$S \Rightarrow^* \alpha A \beta \quad (\text{всі } A \in N')$$

Где N' — нетерминали є досяжними).

Визначення 4.32. Нетерминал $A \in N$ назвемо непродуктивним в граматикою $G=(N, T, P, S)$, якщо з A не можна вивести жодного термінального ланцюжка, тобто не існує виводу виду $A \Rightarrow^* Gt, t \in T^*$.

Спочатку визначимо множину продуктивних нетерминалів. Множина продуктивних нетерминалів для заданої КВ-граматики $G=(N, T, P, S)$ легко визначається за допомогою наступних індуктивних визначень.

$$1. Pr_0 = \{A \mid \exists \text{ правило } A \rightarrow t \in P, \text{ що } t \in T^*\}.$$

$$2. Pr_i = \{A \mid \exists \text{ правило } A \rightarrow \gamma \in P, \text{ що } \gamma \in (R_{i-1} \cup T)^*\} \cup Pr_{i-1} \quad (i=1, 2, \dots).$$

1

Оскільки множина N є скінченною, а формула для визначення Pr_i задає монотонне

i

за i відображення, то існує k ($k \geq 0$), що $Pr = Pr$. Інакше кажучи, послідовність Pr ,

0

Pr_1, Pr_2, \dots стабілізується на k -му кроці. Покладемо $Pr = Pr_k$.

Множина UPr непродуктивних нетерміналів задається формулою: $UPr = N \setminus Pr$. За граматику $G = (N, T, P, S)$ будемо граматику $G' = (N', T', P', S')$ таким чином:

1. $N' = (N \cap Pr) \cup \{S\}$
2. $T' = T$
3. $S' = S$
4. $P' = \{A \rightarrow \alpha \in P \mid \alpha \in (Pr \cup T)^*\}$

Очевидною є наступна лема.

Лема 4.4. Для граматики $G' = (N', T', P', S')$ виконуються наступні властивості:

1. $L(G') = L(G)$ (еквівалентність).

2. Для всіх $A \in N' \setminus \{S\}$ існують термінальні ланцюжки, що виводяться з A .

Відзначимо, що аксіома S є спеціальним випадком, тому що вона може бути непродуктивною, і разом з тим вона має належати множині нетермінальних символів (за визначенням породжувачих граматик), але в такому випадку множина правил буде порожньою.

До речі, наведений метод побудови множини продуктивних нетерміналів дозволяє дати відповідь на питання, чи є порожньою мова, що породжується граматику

$G = (N, T, P, S)$?

Відповідь проста: якщо аксіома є продуктивним нетерміналом, то мова непорожня, якщо ж аксіома - непродуктивний нетермінал, то мова - порожня.

Визначення 4.33. Символ $X \in N \cup T$ назвемо несуттєвим у KB-граматиці $G = (N, T, P, S)$, якщо в ній немає виводу виду $S \Rightarrow^* wXy \Rightarrow^* wxu$, де w, x, u належать T^* .

З наведеного визначення бачимо, що нетермінал X - несуттєвий, якщо X є недосяжним або непродуктивним нетерміналом (крім, можливо, аксіоми). Термінальний символ X є несуттєвим, коли немає жодного породжуваного ланцюжка, що містить X .

Визначення 4.34. Граматика $G = (N, T, P, S)$ називається зведеною, якщо в ній немає несуттєвих символів (можливо, крім аксіоми S).

Для побудови зведеної граматики спочатку видаляємо недосяжні та непродуктивні нетермінали. Потім з множини термінальних символів видаляємо несуттєві символи. Такі термінальні символи не містяться в правих частинах отриманих правил. Отримана граматика буде зведеною.

Лема 4.5. За кожною KB-граматику можна побудувати еквівалентну їй зведену граматику, що не містить несуттєвих символів.

Приклад 4.19. Розглянемо граматику $G = (\{S, A, B, C\}, \{a, b, c\}, P, S)$, де P складається з правил:

- $S \rightarrow a$
- $S \rightarrow A$
- $A \rightarrow AB$
- $B \rightarrow b$

$C \rightarrow c$

Спочатку видаляємо недосяжні нетерміналі. Отримаємо $R = \{S, A, B\}$. Недосяжним є C . З граматики треба видалити вказаний символ та останнє правило. Далі

визначаємо множину продуктивних нетерміналів. Знаходимо, що $Pr = \{S, B\}$. Непродуктивним є A . Після видалення відповідних правил нетермінал B стає недосяжним. Видаляємо і його. Залишається одне правило $S \rightarrow a$. Тому суттєвим термінальним символом є лише a . Видаляємо несуттєві термінальні символи. Отримуємо наступну зведену граматику: $G' = (\{S\}, \{a\}, \{S \rightarrow a\}, S)$.

4.9.2. Видалення $\epsilon\epsilon$ -правил

Визначення 4.35. Назвемо КВ-граматику $G = (N, T, P, S)$ граматику без ϵ -правил (або нескорочуваною), якщо P не містить ϵ -правил, тобто правил виду $A \rightarrow \epsilon$.

Лема 4.6 За кожною КВ-граматику можна побудувати еквівалентну їй (з точністю до порожнього ланцюжка - ϵ -еквівалентну) граматику без ϵ -правил.

Доведення. Нехай дана КВ-граматика $G = (N, T, P, S)$, що породжує мову $L(G)$.

Проведемо серію перетворень множини P . Якщо множина P містить правила $B \rightarrow \alpha A \beta$ і $A \rightarrow \epsilon$ для деяких $A, B \in N, \alpha, \beta \in (N \cup T)^*$, то додамо правило $B \rightarrow \alpha \beta$ в P . Повторюємо цю процедуру поки множина правил не стабілізується. Далі виключимо із множини P всі правила вигляду $A \rightarrow \epsilon$.

Наведений алгоритм можна уточнити за допомогою наступних індуктивних визначень.

1. $P_0 = P$
 2. $P_i = \{B \rightarrow \alpha \beta \mid \exists \text{ правила } B \rightarrow \alpha A \beta \in P_{i-1}, A \rightarrow \epsilon \in P_{i-1}\} \cup R_i$
- ($i=1, 2, \dots$).

Оскільки множина правил є скінченною, а праві частини нових правил коротші, то формула для визначення P_i задає монотонне за i відображення. Тому існує k ($k \geq 0$),

$P_i = P_{i-1}$

що $P_k = P_{k+1}$. Інакше кажучи, послідовність P_0, P_1, P_2, \dots стабілізується на k -му кроці.

Покладемо $P' = P_k \setminus \{A \rightarrow \epsilon \in P_k\}$.

Одержана граMATика $G' = (N, T, P', S)$ породжує мову $L(G) \setminus \{\epsilon\}$.

Дійсно, кожен вивід у граматиці G , який не породжує порожнє слово, може бути промодельований у граматиці G' , що легко доводиться індукцією по довжині виводу.

Так само можна промодельувати виводи в G' виводами в G , використовуючи замість модифікованих правил виду $B \rightarrow \alpha \beta$ початкові правила виду $B \rightarrow \alpha A \beta$ з подальшим виводом порожнього ланцюжка з A . ■

Приклад 4.20. Розглянемо граматику G з правилами

$S \rightarrow aSb$
 $S \rightarrow bSa$
 $S \rightarrow \epsilon$

Застосувавши до цієї граматики описаний метод видалення ϵ -правил, отримаємо граматику G' з правилами

$S \rightarrow aSb$
 $S \rightarrow bSa$

$S \rightarrow ab$
 $S \rightarrow ba,$

яка еквівалентна з точністю до порожнього ланцюжка граматиці G .

4.9.3. Нормальна форма Хомського

Визначення 4.36. Назвемо КВ-граматику $G=(N, T, P, S)$ граматику у нормальній формі Хомського, якщо її правила мають вигляд $S \rightarrow \varepsilon, A \rightarrow a, A \rightarrow BC$ для деяких $A, B, C \in N, a \in T$.

Лема 4.7. За кожною КВ-граматику можна побудувати еквівалентну їй граматику у нормальній формі Хомського.

Доведення. Нехай дана КВ-граматика $G=(N, T, P, S)$. Проведемо ряд перетворень цієї граматики так, що породжувана нею мова залишиться незмінною. Спочатку

побудуємо еквівалентну граматику $G'=(N', T', P', S')$ без ε -правил.

Далі замінимо у всіх правилах кожен термінальний символ a на новий

нетермінальний символ $N(a)$ і додамо до множини P правила $N(a) \rightarrow a$ для всіх $a \in T$.

Видалимо правила вигляду $A \rightarrow A_1 A_2 \dots A_n$, де $n > 2$, замінюючи його на наступний ряд

нових правил $A \rightarrow A_1 N(A_2 \dots A_n), N(A_1 A_2 \dots A_n) \rightarrow A_1 N(A_2 \dots A_n), \dots, N(A_1 A_2 \dots A_n) \rightarrow A_1 A_2 \dots A_n$ (при $n \geq 2$)

цьому додаються нові нетермінальні символи $N(A_1 A_2 \dots A_n), N(A_1 A_2 \dots A_n), \dots, N(A_1 A_2 \dots A_n)$.

Якщо для деяких $A \in N, B \in N$ і $\alpha \in (NUT)^*$ множина P містить правила $A \rightarrow B, B \rightarrow \alpha$,

але не містить правила $A \rightarrow \alpha$, то додамо це правило в P . Повторюємо цю процедуру, доки можливо. Після цього видалимо із множини P всі правила вигляду $A \rightarrow B$.

Нарешті, змінимо S на S' , та додамо правила $S' \rightarrow \varepsilon, S \rightarrow S'$ в тому випадку, коли мова $L(G)$ містить порожній ланцюжок.

Приклад 4.21. Граматика $S \rightarrow \varepsilon, S \rightarrow abSc$ еквівалентна наступній граматиці в нормальній формі Хомського: $S \rightarrow \varepsilon, S \rightarrow AB, B \rightarrow CD, D \rightarrow SE, C \rightarrow b, A \rightarrow a, E \rightarrow c$.

4.9.4. Нормальна форма Грейбах

Визначення 4.37. КВ-граматику $G=(N, T, P, S)$ будемо називати граматику в

нормальній формі Грейбах, якщо її правила мають вигляд $A \rightarrow a\alpha$ ($a \in T, \alpha \in (NUT)^*$),

тобто кожне правило починається з термінального символу.

Сформулюємо без доведення наступне твердження.

Лема 4.8 За кожною КВ-граматику можна побудувати ε -еквівалентну їй (з точністю до порожнього ланцюжка) граматику у нормальній формі Грейбах.

Зауважимо, що наявність нормальної форми Грейбах дозволяє досить легко за кожною граматику побудувати еквівалентний їй недетермінований магазинний автомат. Також у таких граматиках довжина виводу термінального ланцюжка не перевищує його довжину.

Приклад 4.22. Граматика $S \rightarrow \varepsilon, S \rightarrow abSc$ еквівалентна наступній граматиці в нормальній формі Грейбах: $S \rightarrow abc, S \rightarrow abSc$.

4.9.5. Рекурсивні нетермінали

Визначення 4.38. Нетермінал $A \in N$ KB-граматики назвемо рекурсивним (самовставним, циклічним), якщо існує вивід виду $A \Rightarrow^* \alpha A \beta$. Якщо такого виводу немає, то нетермінал називають нерекурсивним.

Лема 4.9. Якщо KB-граматика G не має рекурсивних нетерміналів, то мова $L(G)$ скінченна.

Дійсно, в цьому випадку може бути лише скінченна кількість виводів, тому породжується лише скінченна мова.

Зауважимо, що зворотне твердження не є справедливим для довільної граматики, бо може бути рекурсивний нетермінал, що породжує скінченну мову. Це можливо

для рекурсивного нетерміналу, коли є вивід $A \Rightarrow^* \alpha A \beta$, але завжди з α та β

породжуються порожні ланцюжки: $\alpha \Rightarrow^* \varepsilon$ та $\beta \Rightarrow^* \varepsilon$. Крім того, рекурсивні недосяжні або непродуктивні рекурсивні нетермінали не впливають на породжувану мову.

Приклад 4.23. Розглянемо граматику, множина P якої складається з наступних правил виведення:

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow C \\ A &\rightarrow a \\ B &\rightarrow b \\ C &\rightarrow A \end{aligned}$$

Мова, породжена цією граматиною, складається з єдиного ланцюжка ab . Але нетермінали A та C є рекурсивними.

Отже, наведена лема дозволяє сформулювати необхідну умову породження KB-граматиною нескінченних мов: нескінченна мова може породжуватися лише граматиною з рекурсивними нетерміналами. Але ця умова, як показує приклад, не є достатньою.

Для нескінченних мов має місце наступне твердження.

Лема 4.10. Нехай граматика $G=(N, T, P, S)$ породжує нескінченну мову. Тоді існує суттєвий рекурсивний нетермінал A такий, що має місце $A \Rightarrow^* t A t$, де $t, t \in T^*$ та

$$|t_1 t_2| \geq 1$$

$$|t_1 t_2| \geq 1$$

Ця лема буде використовуватись для доведення властивостей KB-мов.

4.10. Властивості контекстно-вільних мов

У цьому розділі розглянемо властивості KB-мов, тобто структуру їх ланцюжків, виразну потужність KB-мов та їх замкненість відносно операцій над формальними мовами.

Спочатку сформулюємо для КВ-мов аналог твердження (Лема 4.10), яке характеризувало властивості КВ-граматик.

Лема 4.11 (лема про розростання, лема про накачку). Нехай L – КВ-мова над алфавітом T . Тоді знайдеться таке натуральне число k , що для довільного ланцюжка

$t \in L$ довжини не менше k знайдуться ланцюжки $u, v, t_1, t_2, x \in T^*$, для яких вірно

$$t_1 t_2 \dots t_i t_{i+1} \dots t_n = u v t_1 t_2 \dots t_i t_{i+1} \dots t_n v \in L \text{ для всіх } i=0,1,\dots$$

Ідея доведення полягає в тому, що для породження мови L розглядається граматика у нормальній формі Хомського (нескорочуюча граматика). Тоді для достатньо довгих виводів можна виділити рекурсивний нетермінал A такий, що має місце $A \Rightarrow^* t_1 A t_2$, де $t_1, t_2 \in T^*$ та $|t_1 t_2| \geq 1$ (дивись попередню лему). Звідси t буде

$$t_1 t_2 \dots t_1 t_2 \dots t_1 t_2$$

впливати твердження леми.

4.11. Операції над формальними мовами

Лема 4.12. Мова $L_3 = \{a^n b^n c^n \mid n \geq 0\}$ не є КВ-мовою.

Доведення. Якби мова L_3 була б КВ-мовою, то тоді існували б ланцюжки u, v, t_1, t_2 ,

$$t_1 t_2 \dots t_i t_{i+1} \dots t_n = u v t_1 t_2 \dots t_i t_{i+1} \dots t_n v \in L_3 \text{ для всіх } i=0,1,\dots$$

Зрозуміло, що t_1 та t_2 не може складатися з різних символів (інакше для деякого i ланцюжок $u t_1 x t_2 v$ не належав би до L_3).

Нехай $t_1 = a^i$ та $t_2 = b^i$. Тоді ланцюжок $u t_1 x t_2 v$ не може складатися з різних символів (інакше для деякого i ланцюжок $u t_1 x t_2 v$ не належав би до L_3).

$$t_1 t_2$$

буде належати L_3 . Але якщо t_1 складається лише з одного символу, то збільшуючи i , можна порушити баланс символів a, b, c . Тому мова L_3 не може бути КВ-мовою.

Далі почнемо розгляд властивостей КВ-мов відносно теоретико-множинних операцій.

Лема 4.13. Клас КВ-мов замкнений відносно об'єднання.

Дійсно, нехай КВ-мови L_1 та L_2 породжені граматиками $G_1 = (N_1, T, P_1, S_1)$ та $G_2 = (N_2, T, P_2, S_2)$

$$G = (N_1 \cup N_2, T, P_1 \cup P_2, S_1 \cup S_2)$$

відповідно. Задамо граматиками G_1 та G_2 відповідно. Задамо граматиками G_1 та G_2 відповідно.

$$G_1 = (N_1, T, P_1, S_1) \text{ та } G_2 = (N_2, T, P_2, S_2)$$

граматик G_1 та G_2 , наступним чином:

1 2

• перейменуємо (якщо це потрібно) множини нетерміналів N
і N так, щоб

1 2

$N \cap N = \emptyset$, та додамо новий нетермінал S ;
1 2

• об'єднаємо множини правил P і P , додаючи два нових правила
 $S \rightarrow S$ та

1

1 2

$S \rightarrow S_2$.

Таким чином, отримуємо граматику $G = (N \cup N \cup \{S\}, T, P \cup P \cup \{S \rightarrow S, S \rightarrow S\}, S)$,
яка

1 2

1 2

1 2

складається з усіх правил граматик G та G з перейменованими
множинами

1 2

нетерміналів та з доданих правил $S \rightarrow S$, $S \rightarrow S$. З побудови очевидно, що G породжує
1 2

всі слова з мов L та L . Доведемо, що вона не породжує інших слів.
1 2

Згідно твердження (Лема 4.2) перейменування
нетерміналів не змінює
породжувану мову. Оскільки найпершим правилом у виводі є $S \rightarrow S$ або $S \rightarrow S$,
то

1 2

далі ми продовжуємо вивід або для S , або для S , породжуючи слова лише з мови
 L

1 2

1

або L_2 відповідно. Отже, L належить класу КВ-мов.

Лема 4.14. Клас КВ-мов не замкнений відносно перетину.

Для доведення цього факту достатньо розглянути приклад.

Візьмемо мови $L_1 = \{a^n m^m \mid n, m \geq 0\}$ та $L_2 = \{a^n b^n c^m \mid n, m \geq 0\}$. Ці

мови є КВ-мовами.

Однак мова $L \cap L = \{a^n b^n c^n \mid n \geq 0\} = L_3$ не є КВ-мовою.
1 2

Лема 4.15. Клас КВ-мов не замкнений відносно доповнення.

Твердження леми впливає з попередньої леми, оскільки, в силу
законів де

Моргана, будь-який клас мов, замкнений відносно об'єднання та
доповнення, має

бути замкненим відносно перетину. Тобто з припущення замкненості класу КВ-мов
відносно доповнення впливає замкненість відносно перетину, що
суперечить

попередній лемі.

Приклад 4.24. Побудуємо КВ-граматику, яка задає мову, доповнення до якої не буде КВ-мовою.

Ідея побудови впливає з лем 4.14 та 4.15. Візьмемо мову $L_3 = \{a^n b^n \mid n \geq 0\}$, яка не

є КВ-мовою. Покажемо, що її доповнення - мова L_3 є КВ-мовою. Оскільки

$L_3 = L_1 \cap L_2 = L_1 \cup L_2$, то спочатку доведемо, що доповнення мов L_1 та L_2 , тобто мови

L_1 та L_2 , є КВ-мовами. Дійсно, $L_1, L_2 \subseteq a^*b^*c^*$. Остання мова є регулярною, тому її

доповнення $a^*b^*c^* \setminus (a^*b^*c^*) = (\{a, b, c\}^* \setminus a^*b^*c^*)$ також є регулярною мовою. Ця мова

містить слова, які мають заборонені комбінації двох символів, тобто комбінації, які

порушують порядок слідування символів у мовах L_1 та L_2 . Порядок слідування є

наступним:

- літера a передує літері b , або літері c ;
- літера b передує літері c .

Заперечення цих тверджень дає наступні комбінації: ba, ca, cb . Тому $(\{a, b, c\}^* \setminus a^*b^*c^*) = \{a, b, c\}^* ba \{a, b, c\}^* \cup \{a, b, c\}^* ca \{a, b, c\}^* \cup \{a, b, c\}^* cb \{a, b, c\}^*$. Ця

мова задається наступною граматикою:

$S \rightarrow R \mid baR \mid caR \mid cbR$
 $R \rightarrow \varepsilon \mid aR \mid bR \mid cR$

Отже, $L_1 = \{a, b, c\}^* \setminus L_1 = (\{a, b, c\}^* \setminus a^*b^*c^*) \cup (a^*b^*c^* \setminus L_1)$. Покажемо, що мова $(a^*b^*c^* \setminus L_1)$

є КВ-мовою. В ланцюжках цієї мови кількість літер b не співпадає з кількістю літер c .

Ця мова задається граматикою:

$S \rightarrow ABbE$
 $S \rightarrow AEcC$
 $E \rightarrow \varepsilon \mid bEc$
 $A \rightarrow \varepsilon \mid aA$
 $B \rightarrow \varepsilon \mid bB$
 $C \rightarrow \varepsilon \mid cC$

Аналогічно будується граматика другої мови L_2

$= \{a, b, c\}^* \setminus L_2 = (\{a, b, c\}^* \setminus a^*b^*c^*) \cup$

$(a^*b^*c^* \setminus L_2)$, в ланцюжках якої кількість літер a не співпадає з кількістю літер b :

$S \rightarrow QbBC$
 $S \rightarrow AaQC$
 $Q \rightarrow \varepsilon \mid aQb$
 $A \rightarrow \varepsilon \mid aA$
 $B \rightarrow \varepsilon \mid bB$
 $C \rightarrow \varepsilon \mid cC$

Об'єднуючи три граматика (а це можна зробити, бо колізії нетерміналів не

відбувається), отримуємо наступну граматикою:

$S \rightarrow R \mid baR \mid caR \mid cbR \mid ABbE \mid AEcC \mid QbBC \mid AaQC$
 $R \rightarrow \varepsilon \mid aR \mid bR \mid cR$
 $E \rightarrow \varepsilon \mid bEc$
 $Q \rightarrow \varepsilon \mid aQb$
 $A \rightarrow \varepsilon \mid aA$
 $B \rightarrow \varepsilon \mid bB$
 $C \rightarrow \varepsilon \mid cC$

Ця граматика породжує КВ-мову, але її доповнення не є КВ-мовою.

■

Розглянемо тепер деякі специфічні операції над мовами, що відповідають більш конкретному рівню абстракції, який трактує ланцюжки як послідовності символів.

Лема 4.16. Клас KB-мов замкнений відносно конкатенації.

Дійсно, нехай KB-мови L_1 та L_2 породжені граматиками $G_1 = (N_1, T, P_1, S_1)$ та

$G_2 = (N_2, T, P_2, S_2)$ відповідно. Будемо вважати, що $N_1 \cap N_2 = \emptyset$.

Візьмемо символ $S \notin N_1 \cup N_2$. Тоді мова $L = L_1 \cdot L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}$ буде породжена KB-

граматикою $G = (N_1 \cup N_2 \cup \{S\}, T, P_1 \cup P_2 \cup \{S \rightarrow S_1 S_2\}, S)$. Це випливає з того, що вивід будь-якого

ланцюжка починається з доданого правила $S \rightarrow S_1 S_2$, а далі будується з правил P_1 та

P_2 , які виводять лише слова з мов L_1 та L_2 відповідно.

Лема 4.17. Клас KB-мов замкнений відносно ітерації.

Дійсно, нехай KB-мова L породжена граматикою $G = (N, T, P, S)$. Тоді мова L^* буде

породжена KB-граматикою $G = (N \cup \{S'\}, T, P \cup \{S' \rightarrow S'S, S' \rightarrow \varepsilon\}, S')$, де $S' \notin N$. Це впливає з

того, що вивід будь-якого слова починається з доданого правила $S' \rightarrow S'S$, що дозволяє породжувати будь-яку кількість ланцюжків з мови L .

Лема 4.18. Клас KB-мов замкнений відносно дзеркального відображення (обернення) ланцюжків.

Дійсно, нехай KB-мова L породжена граматикою $G = (N, T, P, S)$. Тоді мова L буде

породжена KB-граматикою $G = (N, T, P, S)$, де $P = \{A \rightarrow \alpha \mid A \rightarrow \alpha^R\}$.

Визначимо операції дублювання та дзеркального дублювання наступним чином:

$D(L) = \{ww \mid w \in L\}$ та $DM(L) = \{w\bar{w} \mid w \in L\}$.

Лема 4.19. Клас KB-мов не замкнений відносно операцій дублювання та дзеркального дублювання.

Для доведення цього факту достатньо розглянути приклад. Візьмемо KB-мову

~

породжена KB-граматикою $G = (N, T, P, S)$, де $P = \{A \rightarrow \alpha \mid A \rightarrow \alpha^R\}$.

Визначимо операції дублювання та дзеркального дублювання наступним чином:

$D(L) = \{ww \mid w \in L\}$ та $DM(L) = \{w\bar{w} \mid w \in L\}$.

Лема 4.19. Клас KB-мов не замкнений відносно операцій дублювання та дзеркального дублювання.

Для доведення цього факту достатньо розглянути приклад. Візьмемо KB-мову

$2n$ n n n n n n

$L = \{a^n b^n \mid n \geq 0\}$. Тоді мови $D(L) = \{a^n b^n \mid n \geq 0\}$ та $DM(L) = \{a^n b^n \mid n \geq 0\}$ не можуть бути

KB-мовами в силу леми про розростання.

Таким чином, клас KB-мов не утворює підалгебру теоретико-множинної алгебри формальних мов, бо не є замкненим відносно перетину та доповнення. Тому для KB-мов розглядаються інші алгебри, в першу чергу алгебра $ACF = \{CF, U, \cdot\}$ з операціями об'єднання та конкатенації. Алгебри з такими операціями будемо називати слабкими алгебрами формальних мов (САФМ). Виявляється, цих операцій достатньо для подання KB-мов за допомогою рівнянь у цій алгебрі. Цей факт буде розглянуто пізніше.

4.12. Древа виводу

Визначення 4.39. Нехай $G = (N, T, P, S)$ - контекстно-вільна граматика

$S \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \dots \Rightarrow \alpha$ - вивід у G . Будемо називати цей вивід лівостороннім,

якщо для кожного $i, 0 \leq i < n$, можемо записати α у вигляді $w_i \beta_i$, де

$A_i \rightarrow \gamma_i \in P, w_i \in T^* \text{ і } A_i \in N$. Змістовно вивід є лівостороннім, якщо на кожному кроці

заміняється найлівіший нетермінал.

Очевидним є наступне твердження.

Лема 4.20. Нехай $G = (N, T, P, S)$ - контекстно-вільна граматика. Якщо

$w \in L(G)$ то існує лівосторонній вивід w в G .

Визначення 4.40. Для контекстно-вільних граматики кожному виводу вигляду

$S \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \dots \Rightarrow \alpha$ можна співставити скінченне впорядковане дерево, яке

має назву дерева виводу. Вершини дерева відмічені символами алфавіту $N \cup T$.

Корінь дерева відмічено початковим символом S . Якщо у виводі було застосовано

правило $S \rightarrow \alpha$, то кожному символу ланцюжка α , на який замінюється символ S на першому кроці виводу, ставиться у відповідність вершина дерева, i до неї

проводиться дуга із кореня. Отримані таким чином безпосередні нащадки кореня

впорядковані відповідно до їх порядку у ланцюжку α . Для тих із одержаних вершин,

що відмічені символами з множини N , робиться аналогічна побудова і т.ін. Кроною дерева виводу називається слово, отримане із термінальних символів, записаних на вершинах при їх обході зліва-направо.

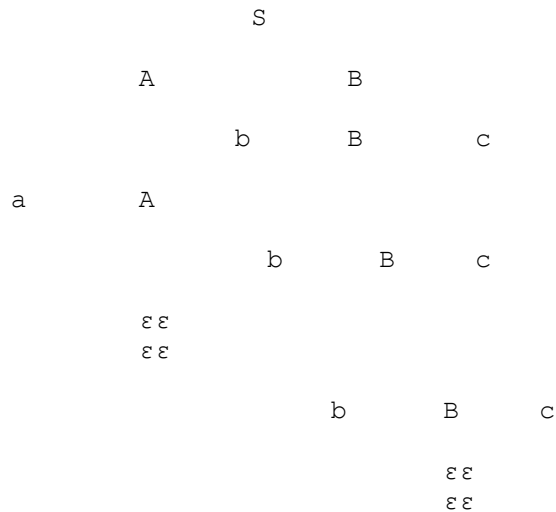
n m m

Приклад 4.25. Візьмемо наступну граматику для мови $L=L_1UL_2$, де $L_1=\{a b c$
|

$n, m \geq 0\}$ та $L_2=\{a^n b^n c^m \mid n, m \geq 0\}$:

$S \rightarrow AB \mid CD \mid \epsilon$
 $A \rightarrow \epsilon \mid aA$
 $B \rightarrow \epsilon \mid bBc$
 $C \rightarrow \epsilon \mid aCb$
 $D \rightarrow \epsilon \mid cD$
Виводу

$S \Rightarrow AB \Rightarrow AbBc \Rightarrow aAbBc \Rightarrow aAbbBcc \Rightarrow aAbbbBccc \Rightarrow abbbBccc \Rightarrow abbbccc$
відповідає наступне дерево виводу:



Неважко довести таку лему.

Лема 4.21. Нехай $G = (N, T, P, S)$ - контекстно-вільна граматики
і $w \in L(G)$.

Тоді існує взаємно-однозначна відповідність між лівосторонніми виведеннями слова
 w в граматиці G і деревами виводу в граматиці G , кроною яких є w .

Дерево виводу фактично задає клас еквівалентності виводів, які
розрізняються

лише порядком застосування правил. Тому для задач, для яких порядок застосувань
не є важливим, доцільно спиратися на дерева виводу. Крім того, дерева
виводу

фактично задають структурну організацію ланцюжка, яка є важливою в
різних

застосуваннях граматик, зокрема, і в семантиці.

4.13. Однозначні та неоднозначні граматики

Визначення 4.41. KB-граматика називається
неоднозначною, якщо існує
ланцюжок, котрий має два або більше різних лівосторонніх виводів.
Інакше KB-
граматика називається однозначною.

Приклад 4.26. KB-граматика із прикладу 4.25 неоднозначна. Слово
 $aabbcc$ має
два різних лівосторонніх вивода:

$S \Rightarrow AB \Rightarrow aAB \Rightarrow aaAB \Rightarrow aaB \Rightarrow aabBc \Rightarrow aabbBcc \Rightarrow aabbcc$
та
 $S \Rightarrow CD \Rightarrow aCbD \Rightarrow aaCbD \Rightarrow aabbD \Rightarrow aabdDc \Rightarrow aabbDcc \Rightarrow aabbcc$

Також неоднозначною є граматика (БНФ) мови SIPL, наведена в першому розділі.

Приклад 4.2. Мова Діка - це мова, що породжена граматиною $G = (\{S\}, T, P, S)$,

к

де $k > 0$, $T = \{ \alpha_1, \alpha_2, \dots, \alpha_k, b_1, b_2, \dots, b_k \}$, а P складається з продукцій $S \rightarrow \alpha_i S b_i$, $S \rightarrow \epsilon$,

$S \rightarrow \alpha_i S b_i$, $1 \leq i \leq k$. Нехай $L = L(G)$. Зрозуміло, що мова Діка є контекстно-вільною

мовою. Можна вважати, що мова Діка - це множина усіх ланцюжків урівноважених дужок k різних типів, де a_i та b_i - це ліва та права дужки для кожного i . Мова Діка є

важливою в теорії контекстно-вільних граматик. Вона, зокрема, утворює основу довільної контекстно-вільної мови, тому що будь-яка КВ-мова є гомоморфним образом мови Діка. Наведена граматика не є однозначною. Але наступна граматика $S \rightarrow \epsilon$, $S \rightarrow a_i S b_i$, $1 \leq i \leq k$ - однозначна.

Визначення 4.42. КВ-мова називається суттєво неоднозначною, якщо кожна КВ-граматика, яка породжує цю мову, є неоднозначною.

n m m

Приклад 4.28. КВ-мова $L = L_1 U L_2$ з прикладу 4.25, де $L_1 = \{ a^n b^m c^m \mid n, m \geq 0 \}$ та

$L_2 = \{ a^n b^n c^m \mid n, m \geq 0 \}$ є суттєво неоднозначною. Доведення цього факту наводиться в книзі [1], стор. 234-236.

Неоднозначність є важливим поняттям теорії мов. Якщо граматика, яка використовується для породження мови програмування, є неоднозначною, то це може призвести до її неоднозначної семантичної інтерпретації, що є небажаним явищем.

4.14. Розв'язні та нерозв'язні проблеми КВ-граматик та мов

Масову проблему називають алгоритмічно розв'язною, або розв'язною, якщо предикат, який її визначає, є рекурсивний, інакше проблему називають алгоритмічно нерозв'язною, або нерозв'язною.

Масову проблему називають частково алгоритмічно розв'язною, або частково розв'язною, або напіврозв'язною, якщо предикат, який її визначає, є частково рекурсивний.

Для КВ-граматик та мов наступні проблеми є розв'язними :

1. Чи є мова, породжувана КВ-граматикою, порожньою?
2. Чи є мова, породжувана КВ-граматикою, скінченною?
3. Чи є мова, породжувана КВ-граматикою, нескінченною?
4. Чи належить ланцюжок в мові, що породжується КВ-граматикою?

Для КВ-граматик та КВ-мов наступні проблеми є нерозв'язними.

1. Чи є перетин мов, породжуваних двома КВ-граматиками, порожнім?
2. Чи є перетин мов, породжуваних двома КВ-граматиками, скінченим?
3. Чи є перетин мов, породжуваних двома КВ-граматиками, нескінченим?
4. Чи є КВ-граматика однозначною?
5. Чи є порожнім (скінченим, нескінченим) доповнення до КВ-мови?
6. Чи співпадає КВ-мова, породжувана граматикою, з T^* ?
7. Чи є еквівалентними дві КВ-граматики?
8. Чи є регулярною мова, породжувана КВ-граматикою?

Зазначимо, що проблеми, розв'язні для КВ-граматик та мов будуть розв'язними

і для регулярних граматик та мов. Разом з тим деякі проблеми, нерозв'язні для КВ-

граматик та мов, можуть бути розв'язними для регулярних граматик та мов.

Зокрема, такими є вищенаведені проблеми 1-7, переформульовані на випадок

регулярних граматик та мов.

4.15. Рівняння в алгебрах формальних мов

Граматики та БНФ є формалізмами, що задають мову «поштучно», «поелементно», вказуючи механізм породження окремих слів. Разом з тим сама форма граматик та БНФ підштовхує до думки, що їх можна розглядати як системи рівнянь, розв'язками яких є мови. Щоб перейти до такого тлумачення, попередньо введемо необхідні визначення.

Визначення 4.43. Слабкою алгеброю формальних мов над алфавітом T будемо

називати алгебру $AL = (2T^*, U,)$ з операціями об'єднання та добутку мов.

Визначення 4.44. Множиною виразів $LExp$ над множиною змінних Var та

множиною мов-констант $LS = \{ L \mid L \subseteq T^*, i \in I \}$ називається множина, задана

наступним індуктивним визначенням:

- 1) якщо $x \in Var$, то $x \in LExp$,
- 2) якщо $L \in LS$, то $L \in LExp$,
- 3) якщо $t, t' \in LExp$, то tUt' , $t \sqcup t' \in LExp$.

Визначення 4.45. Формальним рівнянням (формальною рівністю) над алгеброю

AL називається запис вигляду $t=t'$, де $t, t' \in LExp$.

Визначення 4.46. Інтерпретацією (означуванням, оцінкою) змінних називається довільне відображення $v : Var \rightarrow 2T^*$. Інтерпретація змінних однозначно (та гомоморфно) продовжується до відображення $\mu : LExp \rightarrow 2T^*$ інтерпретації виразів,

v

параметром якого є інтерпретація змінних v , яке задається індуктивно за побудовою

виразу $e \in L_{\text{Exp}}$:

- 1) якщо $e = x$ ($x \in \text{Var}$), то $\mu_v(e) = v(x)$,
- 2) якщо $e = L$ ($L \in \text{LS}$), то $\mu_v(e) = L$,
- 3) якщо $e = t \cup t'$, то $\mu_v(e) = \mu_v(t) \cup \mu_v(t')$,
- 4) якщо $e = t \sqcap t'$, то $\mu_v(e) = \mu_v(t) \sqcap \mu_v(t')$.

Визначення 4.47. Наведене визначення дозволяє абстрагувати відображення μ_v

до оператора $\mu: L_{\text{Exp}} \rightarrow ((\text{Var} \rightarrow 2T^*) \rightarrow 2T^*)$. Вираз e , який містить змінні x_1, \dots, x_n , будемо

позначати $e(x_1, \dots, x_n)$, а відповідний оператор — $\mu(e(x_1, \dots, x_n))$.

Приклад 4.29. Нехай $\text{Var} = \{x, y, z\}$, $v(x) = \{a, ab\}$, $v(y) = \{\varepsilon, b, cc\}$, $v(z) = \{bb, c\}$. Тоді вираз

$xz\{a\}y$ інтерпретується таким чином:

$$\mu_v(xz\{a\}y) = \{a, ab\} \{bb, c\} \{a\} \{\varepsilon, b, cc\} = \{aa, aab, aba, abab\} \{bba, ca\} \{\varepsilon, b, cc\} =$$

$\{aabba, aabbba, ababba, ababbba, aaca, aabca, abaca, ababca\} \{\varepsilon, b, cc\} =$
 $\{aabba, aabbba, ababba, ababbba, aaca, aabca, abaca, ababca, b, cc\} =$
 $\{aabbab, aabbab, ababbab, ababbab, aacab, aabcab, abacab, ababcab,$
 $aabbacc,$
 $aabbacc, ababbacc, ababbacc, aacacc, aabacc, abacacc, ababacc\}.$

Визначення 4.48. Інтерпретація (означування) змінних $v: \text{Var} \rightarrow 2T^*$ називається розв'язком рівняння $t = t'$, якщо $\mu_v(t) = \mu_v(t')$.

Визначення 4.49. Розв'язком системи рівнянь $\{t_1 = t'_1, \dots, t_n = t'_n\}$ є така

інтерпретація змінних, яка кожне рівняння перетворює у рівність.

Визначення 4.50. Рівняння виду $x = t$, де $x \in \text{Var}$, $t \in L_{\text{Exp}}$ називається рекурсивним.

Якщо у виразі t фігурують лише скінченні мови, то рівняння називають слабкорекурсивним. Аналогічно вводяться поняття рекурсивних та слабкорекурсивних систем рівнянь.

Розв'язок рівнянь в алгебрах мов можна знаходити різними методами, але найважливішим є метод поступових наближень.

Приклад 4.30

Розглянемо мову $L = \{a^n b^n \mid n \geq 0\}$. Ця мова породжується

наступною простю

граматикою $G: S \rightarrow \varepsilon \mid aSb$.

Цій граматиці відповідає наступне рівняння: $S = \{\varepsilon\} \cup \{a\} S \{b\}$. Позначимо оператор

$\mu(\{\varepsilon\} \cup \{a\} S \{b\})$ як $\varphi(S)$.

Розв'язок рівняння знаходять методом послідовних наближень. Найперше

наближення - порожня мова \emptyset . Наступні наближення отримуємо підстановкою в

оператор $\varphi(S)$ замість S попереднього наближення. (Для спрощення тут можна говорити про підставку в саме рівняння). Отримуємо наступну послідовність наближень:

$$\begin{aligned} R(0) &= \emptyset; \\ R(1) &= \{\varepsilon\}; \\ R(2) &= \{\varepsilon\} \cup \{ab\} = \{\varepsilon, ab\}; \\ R(3) &= \{\varepsilon, ab\} \cup \{ab, aabb\} = \{\varepsilon, ab, aabb\}; \\ &\dots \\ R^{(i+1)} &= R^{(i)} \cup \{a\} R^{(i)} \{b\}; \\ &\dots \end{aligned}$$

Вважаємо, що

$$R = \bigcup_{i \in \omega} R^{(i)}.$$

(Тут ω є першим граничним ординаром і може тлумачитись як множина натуральних чисел. Детальніше про це буде сказано у наступному розділі посібника.)

$$R^{(i+1)} = R^{(i)} \cup \{a\} R^{(i)} \{b\}$$

Використовуючи оператор φ , отримуємо, що $R = \varphi(R)$ та $R = \bigcup_{i \in \omega} \varphi(R^{(i)})$. Мова $R^{(i)}$ буде розв'язком нашого рівняння.

Щоб це довести, розглянемо наступну властивість оператора φ .

$$\varphi\left(\bigcup_{i \in \omega} R^{(i)}\right) = \bigcup_{i \in \omega} \varphi(R^{(i)})$$

Доведення

$$\varphi\left(\bigcup_{i \in \omega} R^{(i)}\right) \subseteq \bigcup_{i \in \omega} \varphi(R^{(i)})$$

Спочатку доведемо, що $\varphi\left(\bigcup_{i \in \omega} R^{(i)}\right) \subseteq \bigcup_{i \in \omega} \varphi(R^{(i)})$. Нехай $x \in \varphi\left(\bigcup_{i \in \omega} R^{(i)}\right)$. Тоді $x \in \{\varepsilon\}$, або

$$x \in \{a\} \left(\bigcup_{i \in \omega} R^{(i)}\right) \{b\}.$$

У першому випадку очевидно, що $x \in \bigcup_{i \in \omega} \varphi(R^{(i)})$. У другому випадку

$$\left(\bigcup_{i \in \omega} R^{(i)}\right) \{b\} \subseteq \bigcup_{i \in \omega} \left(R^{(i)} \{b\}\right)$$

$x \in \varphi(R)$, тому $x \in \bigcup_{i \in \omega} \varphi(R^{(i)})$.

(i) $\varphi(R^{(i)}) \subseteq \varphi(R^{(i+1)})$ (i) $\varphi(R^{(i)}) \subseteq \varphi(R^{(i)})$

Тепер доведемо, що $\varphi(\bigcup_{i \in \omega} R^{(i)}) \supseteq \bigcup_{i \in \omega} \varphi(R^{(i)})$. Нехай тепер $x \in \bigcup_{i \in \omega} \varphi(R^{(i)})$.
Тоді існує $k \in \omega$,

(k) $x \in \varphi(R^{(k)})$ (i) $x \in \varphi(R^{(k)})$ (k-1)

що $x \in \varphi(R^{(k)})$. Це означає, що $x \in \{\varepsilon\}$, або $x \in \{a\}(R^{(k-1)})\{b\}$. Тому $x \in \{a\}(\bigcup_{i \in \omega} R^{(i)})\{b\}$, тобто

(i) $x \in \varphi(\bigcup_{i \in \omega} R^{(i)})$.
■

Умова неперервності фактично говорить, що $R = \bigcup_{i \in \omega} R^{(i)}$

$i \in \omega \varphi(R^{(i)})$ є розв'язком рівняння

(i+1) $\varphi(R^{(i+1)}) \subseteq \varphi(R^{(i+1)})$ (i) $\varphi(R^{(i+1)}) \subseteq \varphi(R^{(i+1)})$ (i) $\varphi(R^{(i+1)}) \subseteq \varphi(R^{(i+1)})$
 $S = \{\varepsilon\} \cup \{a\} S \{b\}$. Дійсно, для $i \in \omega$ маємо, що $\varphi(R^{(i+1)}) = R^{(i+1)}$,
тому $\bigcup_{i \in \omega} \varphi(R^{(i+1)}) = \bigcup_{i \in \omega} R^{(i+1)} = \bigcup_{i \in \omega} R^{(i)} = R$.

Остання рівність випливає з того, що $R^{(0)} = \emptyset$, тому маємо, що $R = \varphi(R)$.

Неважко довести, що цей розв'язок співпадає з мовою, породженою граматикою

G.

Лема 4.23. $L(G) = R$.

Доведення проводиться індукцією. Індуктивне твердження наступне: нехай $L(G) \cap R^k$

k

– усі термінальні слова, що виводяться з S виводами довжини не більше k , тоді
 $L(G) \cap R^k = R^k$.

0 (0)

База індукції. Для $k=0$ маємо $L(G) \cap R^0 = R^0 = \emptyset$.

Крок індукції. Нехай індуктивна гіпотеза справедлива для довільного k .

Доведемо

її справедливість для $k+1$.

Дійсно, виводи довжини не більше ніж $k+1$ отримуємо або застосуванням
правила $S \rightarrow \varepsilon$ (тоді гіпотеза справедлива), або з виводів довжини не більш ніж k
після застосування правила $S \rightarrow aSb$. У цьому випадку, всі породжені ланцюжки
будуть належати $R^{(k+1)}$. І навпаки, якщо ланцюжок належить $R^{(k+1)}$, то він
виводиться не більше ніж за $k+1$ крок.

Доведенням леви завершується розгляд прикладу 4.30.

У наведеному прикладі ми знайшли лише один розв'язок рівняння.
Неважко довести, що цей розв'язок дійсно є єдиним. Доведення можна
провести від

супротивного, вибравши з іншого розв'язку R' найменше за довжиною слово, що не належить R , тоді з рівняння має впливати існування ще меншого слова, що не належить R . Отримане протириччя говорить про єдиність розв'язку. (Деталі доведення з'ясуйте самостійно.)

Виникає питання, чи завжди розв'язок має бути єдиним. Рівняння $S=SS$ є прикладом того, що це не так, бо воно має безліч розв'язків. Наприклад, його

розв'язками є такі мови: $S=\emptyset$, $S=\{\varepsilon\}$, $S=\{a\}^*$, $S=\{aa\}^*$, $S=\{ab\}^*$, $S=\{aba\}^*$ і т.ін.

Інші питання, пов'язані з розв'язком рекурсивних рівнянь, розглянемо у наступному розділі, присвяченому рекурсії.

Висновки

У цьому розділі було розглянуто основні методи подання синтаксису програм. Серед таких методів можна виділити методи породження та сприйняття мов. Формалізмами породження мов є породжуючі граматики, сприйняття мов – автомати (машини). Синтаксис мов програмування звичайно подається формалізмами, які еквівалентні контекстно-вільним грамадикам. У розділі визначено

класифікацію граматик за Хомським, зв'язок класів граматик з класами автоматів, та проведено детальне дослідження класу контекстно-вільних граматик.

Викладені у цьому розділі питання розглядаються у книжках [1, 5, 9].

Основні результати цього розділу полягають у наступному:

1. Побудовано над-абстрактну та абстрактну моделі формальних мов як множин речень.
2. Побудовано дескриптивні формалізми визначення мов як транзиційних систем та породжуючих граматик.
3. Дані визначення основних понять формальних мов та породжуючих граматик: відношення безпосереднього виводу, його рефлексивного транзитивного замикання, виводу слова, породженої мови.
4. Наведено приклад доведень у теорії породжуючих граматик.
5. Наведено класифікацію класів граматик за Хомським.
6. Дано визначення загально-контекстних граматик.
7. Наведено визначення різних формалізмів сприйняття мов.
8. Встановлено еквівалентність між класами граматик та відповідними класами автоматів (машин).
9. Розглянуто методи подання синтаксису мов програмування.
10. Розглянуто властивості контекстно-вільних граматик та мов.
11. Сформульовано розв'язні та нерозв'язні проблеми породжуючих граматик та мов.
12. Розглянуто рівняння в алгебрах мов.

Контрольні питання

1. Дайте визначення синтаксичного аспекту програм.
2. Від яких аспектів програм абстрагуються при вивченні синтаксичного аспекту? Сформулюйте відповідний принцип.
3. Сформулюйте принцип, який вказує на співвідношення синтаксичного та семантичного аспектів?
4. Поясніть вживання терміну «формальна мова».

5. Обґрунтуйте принцип теоретико-множинного тлумачення формальних мов.
Порівняйте цей принцип з теоретико-функціональним принципом формалізації програм.
6. Як визначається над-абстрактна модель формальної мови?
7. Як визначається абстрактна модель формальної мови?
8. Обґрунтуйте конкретизацію речень як послідовностей символів.
Сформулюйте відповідний принцип.
9. Обґрунтуйте необхідність введення дескриптивної компоненти у модель формальної мови.
10. Обґрунтуйте вибір транзиційної системи як дескриптивної моделі формальних мов. Сформулюйте відповідний принцип.
11. Обґрунтуйте необхідність подальшої конкретизації транзиційних систем. У чому суть такої конкретизації?
12. Вкажіть відображення абстракції та конкретизації для чотирьох моделей формальної мови. Намалюйте відповідну таблицю.
13. Дайте визначення понять алфавіту, ланцюжка, порожнього ланцюжка, підланцюжка, операцій конкатенації та піднесення до степені.
14. Що таке вільна напівгрупа?
15. Що таке формальна мова?
16. Визначте теоретико-множинні операції над формальними мовами.
17. Що таке операція конкатенації (добутку) формальних мов?
18. Що таке операція ітерації формальних мов?
19. Що таке операція обернення формальної мови?
20. Що таке породжувача граматики?
21. Дайте визначення відношення безпосередньої вивідності та його рефлексивного транзитивного замикання.
22. Дайте визначення виводу в граматиці.
23. Що таке словоформа (сентенційна форма)?
24. Дайте визначення мови, що породжується граматикою.
25. Вкажіть на екстенціональні та інтенціональні аспекти формальних мов та породжувачих граматик.
26. Визначте класи граматик за Хомським. Яке співвідношення цих класів?
27. Які граматики називаються загально-контекстними?
28. Як співвідносяться класи породжувачих та загально-контекстних граматик?
29. У чому полягає відмінність між механізмами породження та сприйняття мов?
30. Які особливості вирізняють автомати від граматик?
31. Дайте визначення машини Тьюрінга. Вкажіть на інтенціональні та екстенціональні аспекти для формалізму машин Тьюрінга. Як визначається мова, яка розпізнається машиною Тьюрінга?
32. Як співвідносяться машини Тьюрінга та породжувачі граматики?
33. Дайте визначення лінійно-обмеженого автомата.
34. Як співвідносяться лінійно-обмежені автомати та породжувачі граматики?
35. Дайте визначення магазинного автомата.
36. Як співвідносяться магазинні автомати та породжувачі граматики?
37. Дайте визначення скінченного автомата.
38. Як співвідносяться скінченні автомати та породжувачі граматики?
39. Дайте визначення регулярної мови.
40. Як пов'язані регулярні мови з породжувачими граматиками та автоматами?
41. Які методи подання синтаксису використовуються для мов програмування?
42. Що таке нормальні форми Бекуса-Наура?

43. Які конструкції вживаються для модифікованих БНФ?
 - . Що таке синтаксичні діаграми?
45. Як зв'язані БНФ, модифіковані БНФ, контекстно-вільні граматики та синтаксичні діаграми?
46. Сформулюйте основні властивості КВ-граматик.
47. Що таке продуктивні та досяжні нетерміналі?
48. Яка граMATика називається зведеною?
49. Дайте визначення різним нормальним формам КВ-граматик.
50. Що таке рекурсивний нетермінал?
51. Сформулюйте основні властивості КВ-мов.
52. Сформулюйте властивості замкненості КВ-мов відносно основних операцій.
53. Що таке дерево виводу в КВ-граматиці?
54. Які граматики називаються однозначними?
55. Які мови називаються суттєво неоднозначними?
56. Сформулюйте розв'язні проблеми КВ-мов та граматик.
57. Сформулюйте нерозв'язні проблеми КВ-мов та граматик.
58. Що таке алгебра мов?
59. Що таке рівняння в алгебрі мов?
60. Яким чином знаходяться розв'язки рівнянь в алгебра мов?

Вправи

1. Побудуйте приклади транзиційних систем і мов, які вони задають.
2. Побудувати граматику G для мови $L = \{a^n b^{2n} c^n \mid n > 0\}$, довести, що $L(G) = L$, та що мова L не є КВ мовою.
3. Побудувати граматику G і систему рівнянь S для мови $L = \{a^n b^n \mid n > 0\}$ та довести, що $L(G) = L$ та $R(S) = L$.
4. Перетворіть граматику

$$\begin{aligned} S &\rightarrow A \mid B \\ A &\rightarrow aB \mid bS \mid b \\ B &\rightarrow AB \mid Ba \\ B &\rightarrow AS \mid b \end{aligned}$$
 на еквівалентну їй КВ-граматику, що не містить несуттєвих символів.
5. Побудуйте граматику без ε -правил, еквівалентну даній

$$\begin{aligned} S &\rightarrow ABC \\ A &\rightarrow BB \mid \varepsilon \\ B &\rightarrow CC \mid a \\ C &\rightarrow AA \mid b \end{aligned}$$
6. Опишіть мову, яка породжується правилами

$$\begin{aligned} S &\rightarrow bSS \\ S &\rightarrow a \end{aligned}$$
7. Побудуйте, якщо це можливо, КВ-граматики, які породжують наступні мови:
 - а) $\{a^n * n \mid n \geq 1\}$;
 - б) $\{\alpha \mid \alpha \in \{a, b, c\}^*, |\alpha| = |\alpha| = |\alpha|\}$;
 - в) $\{ww \mid w \in \{a, b\}^*\}$;
 - г) $\{a^m n^m \mid m \geq 1, n \geq 1\}$.
8. Для вищенаведених мов побудуйте відповідні автомати, які розпізнають ці мови.

5. Теорія рекурсії (теорія найменшої нерухомої точки)

5.1. Рекурсивні визначення та рекурсивні рівняння

Рекурсивні визначення - це такі визначення, в правій частині яких використовується посилання на поняття, що визначається. Такі визначення мають вид

$$x = \Phi(x).$$

Рекурсивне визначення можна тлумачити

- операційно, тобто вказати алгоритм, за яким можна обчислити рекурсивно визначений об'єкт;
- або денотаційно, тобто як рівняння, розв'язком якого є нерухомі точки

(НТ) оператора Φ .

Зауваження 5.1. Тут на $\Phi(x)$ ми дивимось синкретично (не розрізняючи різні аспекти), та тлумачимо $\Phi(x)$ або як вираз у певній алгебрі (коли говоримо про рівняння), або як оператор у цій алгебрі (коли говоримо про нерухомі точки оператора).

Зауваження 5.2. Часто також розглядаються системи рівнянь виду

$$\begin{aligned} x_1 &= \Phi_1(x_1, x_2, \dots, x_n) \\ x_2 &= \Phi_2(x_1, x_2, \dots, x_n) \\ &\dots \\ x_n &= \Phi_n(x_1, x_2, \dots, x_n) \end{aligned}$$

Такі системи зводяться до одного рівняння над послідовністю (x_1, x_2, \dots, x_n) .

Історія математики та логіки говорить про необхідність обережного поводження з рекурсією. Розглянемо приклад.

Припустимо, що ми хочемо визначити суму $2+2+2+2+\dots$. Позначимо цю суму

через x , тобто $x=2+2+2+2+\dots$. Якщо винести 2 з усіх членів суми, крім першого члена, отримаємо наступне рекурсивне визначення: $x=2+2x$. Звідси $x = -2$, що зовсім не відповідає очікуванню.

Рекурсія може бути прихованою (неявною). Для ілюстрації розглянемо парадокс

Рассела з теорії множин. А саме, множина x називається нормальною, якщо $x \notin x$.

Позначимо через N множину усіх нормальних множин, тобто $N = \{ x \mid x \notin x \}$.

Парадокс

виявляється, якщо запитати, чи є N нормальною множиною? Отримуємо, що $N \in N$ тоді і тільки тоді, коли $N \notin N$. Тут рекурсія виступає неявно, бо в визначенні N (неявно) припускається, що N може бути елементом N .

Наведені приклади говорять про необхідність детального вивчення рекурсії, щоб уникнути некоректностей та парадоксів.

Рекурсія широко використовується в мовах програмування. У таких випадках вона, як правило, визначається операційно, тобто вказується алгоритм, за яким можна обчислити рекурсивну процедуру або функцію.

Традиційні проблеми, що розглядаються для такого роду рівнянь, є проблеми існування та опису всіх можливих розв'язків, зокрема, формулювання умов єдиності розв'язку.

Існують різні методи розв'язку рекурсивних рівнянь. Найчастіше використовують метод послідовних наближень, який полягає у наступному.

Береться початкове наближення d_0 . Далі обчислюється послідовність наближень

$$d_1 = \varphi(d_0), d_2 = \varphi(d_1), \dots$$

За результат береться границя обчисленої послідовності: $d = \lim_{i \rightarrow \infty} d_i$.

i

$i \in \omega$

Зауваження 5.3. У теорії найменших нерухомих точок зазвичай

використовується позначення виду $i \in \omega$, де ω - перший нескінченний ординал, тобто $\omega = \{0, 1, 2, \dots\}$ - множина натуральних чисел.

Метод послідовних наближень графічно представлений на рис. 5.1. Наближення

d_0, d_1, d_2, \dots мають границю d , яка є коренем рівняння $x = \varphi(x)$.

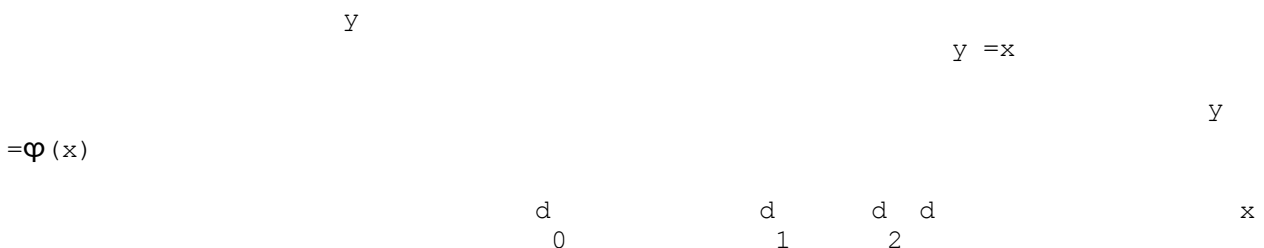


Рисунок 5.1 Розв'язок рівнянь методом послідовних наближень

5.2. Частково впорядковані множини, границі ланцюгів та ω -області

Для того, щоб метод послідовних наближень міг бути застосований, необхідно задати відношення «наближеності» (це – частковий порядок), початкове наближення (у нашому випадку це буде найменший елемент), та гарантувати існування границі (повнота).

Вказані поняття можна формалізувати наступним чином.

Визначення 5.1. Множина D з заданим бінарним відношенням \leq $\subseteq D \times D$ називається частково впорядкованою множиною (D, \leq) (ЧВМ), якщо для відношення \leq виконуються наступні аксіоми часткового порядку ($d, d_1, d_2, d_3 \in D$):

1. 2. 3.

1. Рефлексивність: $d \leq d$.

2. Транзитивність: $d_1 \leq d_2$ & $d_2 \leq d_3 \Rightarrow d_1 \leq d_3$.

3. Антисиметричність: $d_1 \leq d_2$ & $d_2 \leq d_1 \Rightarrow d_1 = d_2$.

Визначення 5.2. Нехай (D, \leq) – ЧВМ, $X = \{d_i\}_{i \in \omega}$ – індексована підмножина D . Ця

$\{d_i\}_{i \in \omega}$

підмножина – ланцюг, якщо $d_0 \leq d_1 \leq d_2 \leq \dots$.

Для частково впорядкованих множин границею ланцюга вважається його точна верхня границя (супремум, найменша мажоранта).

Визначення 5.3. Нехай $X = \{d_i\}_{i \in \omega}$ – ланцюг в ЧВМ (D, \leq) . Границею X

$\{d_i\}_{i \in \omega}$

(позначається $\lim X = \sup X = C X = \lim d_i = \sup d_i = C d_i$) називається точна верхня

$i \in \omega$ $i \in \omega$ $i \in \omega$

границя (супремум, найменша мажоранта) множини X , якщо вона існує.

В теорії рекурсії зазвичай використовується позначення d_i , або $C d_i$, або C_i , або C_i .

C_i $i \in \omega$ i

$i \in \omega$

$C \{d_i \mid i \in \omega\}$, або $C X$. Останнім позначенням i будемо користуватися у подальшому.

i

Визначення 5.4. ЧВМ (D, \leq) – повна, якщо для довільного ланцюга $\{d_i\}_{i \in \omega}$ з D

$i \in \omega$

існує його границя $C d_i$ (що належить D).

$i \in \omega$

Інтегруючи наведені визначення, отримуємо поняття ω -області.

Визначення 5.5. Множина D - ω -область (також вживається термін індуктивна множина, ω -домен), якщо

- 1) на D введено частковий порядок \leq ;
- 2) в D існує найменший елемент 1 ;
- 3) D є повною ЧВМ.

5.3. Неперервні відображення

Центральним поняттям теорії рекурсії є поняття неперервного відображення. На відміну від визначення неперервності за Коші (ϵ - δ визначення) будемо користуватися більш абстрактним визначенням неперервності за Гейне.

Визначення 5.6. Відображення $\varphi : D \rightarrow D$, задане на ЧВМ (D, \leq) , - неперервне за Гейне, якщо для будь-якого ланцюга $\{d_i\}_{i \in \omega}$ з D виконується рівність

$$\varphi \left(\sup_{i \in \omega} d_i \right) = \sup_{i \in \omega} \varphi(d_i).$$

Зауваження 5.4. Поняття неперервного відображення узагальнюється для відображень типу $\varphi : D \rightarrow R$, (де (D, \leq) та (R, \leq) - ЧВМ) наступним чином. Нехай

$\{d_i\}_{i \in \omega}$ - ланцюг в D . Тоді φ - неперервне, якщо $\varphi \left(\sup_{i \in \omega} d_i \right) = \sup_{i \in \omega} \varphi(d_i)$.

Визначення 5.7. Відображення $\varphi : D \rightarrow D$ - монотонне, якщо

$$\forall d_0, d_1 \in D : d_0 \leq d_1 \Rightarrow \varphi(d_0) \leq \varphi(d_1).$$

Неперервні відображення є монотонними.

Лема 5.1. Нехай (D, \leq) - ЧВМ та $\varphi : D \rightarrow D$ - неперервне відображення, тоді φ - монотонне.

Доведення

Нехай $\{d_i\}_{i \in \omega}$ є ланцюг $d_0 \leq d_1 \leq d_2 \leq \dots$, всі елементи якого, крім d_0 , дорівнюють d_1 .

Тоді з визначення ланцюга та неперервності φ випливає, що

$$\varphi(d_1) = d_1$$

$$\varphi(Cd) = C\varphi(d) = C\varphi(d) = \varphi(C\{d, d\}) = \varphi(d)$$

$$1 \quad 0 \quad 1 \quad i \quad i \quad i \quad 0$$

$$i \in \omega \quad i = \omega \quad i = 0$$

Оскільки $C\{d, d\} = d$, то $\varphi(d) = \varphi(d) \cup C\varphi(d)$.
 Це означає, що $\varphi(d) = \varphi(d) \cup C\varphi(d)$.

мажорантою $\varphi(d)$. Тому $\varphi(d) \leq \varphi(d)$. ■

5.4. Теореми про нерухомі точки

Центральною теоремою є наступна теорема, яка стверджує наявність найменшого розв'язку рекурсивного рівняння, який може бути знайдений методом послідовних наближень. Цю теорему відносять до розряду фольклорних («народних») теорем, тобто її формулювання всі знають, але першого автора назвати важко. Як правило, до перших авторів відносять Кнастера, Тарського та Кліні. Важливість теорем пов'язана з широким колом її застосувань.

Теорема 5.1 (Кнастер-Тарський-Кліні)

Нехай D - ω -область, $\varphi : D \rightarrow D$ - неперервне відображення, задане на цій області.

Тоді існує найменша нерухома точка φ , яка позначається $\text{lfp}\varphi$ та для якої справедлива наступна формула:

$$(KTK) \quad \text{lfp}\varphi = C\varphi(\perp),$$

$$(0) \quad \varphi(\perp) = \perp, \quad \varphi(\perp) = \varphi(\varphi(\perp)), \quad i \in \omega.$$

Доведення теореми складається з трьох тверджень:

)i

1. Доведення факту, що множина $\{\varphi(\perp)\}_{i \in \omega}$ є ланцюг (тому її супремум

$$C\varphi(\perp) \text{ існує).}$$

2. Доведення того, що $C\varphi(\perp)$ є нерухомою точкою φ .

$$i \in \omega$$

$$()i$$

3. Доведення, що $\sup_{i \in \omega} \varphi(i)$ є найменшою з нерухомих точок φ .

$$i \in \omega$$

Почнемо з першого твердження. Розглянемо послідовність $\varphi(1), \varphi(\varphi(1)), \dots$.

Доведемо методом математичної індукції, що це - ланцюг.

База індукції. Оскільки 1 - найменший елемент D , то $1 \leq \varphi(1)$.

$(1)^i$ Індуктивний крок. Треба довести, що з $\varphi(1) \leq \varphi(1)$ випливає

$(1)^{i+1} \leq \varphi(1)$. Дійсно, за монотонністю φ маємо

$(1)^i \leq \varphi(1) \Rightarrow \varphi((1)^i) \leq \varphi(\varphi(1)) \Rightarrow \varphi(1)^{i+1} \leq \varphi(1)$.

Отже $\varphi(1), \varphi(\varphi(1)), \dots$ - ланцюг.

Перейдемо до другого твердження. За неперервністю φ :

$(1)^i \leq \varphi(1) \Rightarrow \varphi((1)^i) \leq \varphi(\varphi(1)) \Rightarrow \varphi(1)^{i+1} \leq \varphi(1)$.

(0) Остання рівність випливає з того, що $\varphi(1) = 1$, а найменший елемент не

впливає на супремум. Отже, $\sup_{i \in \omega} \varphi(i)$ - нерухома точка φ .

$(1)^i$ Доведемо тепер останнє твердження, що $\sup_{i \in \omega} \varphi(i)$ - найменша нерухома точка

$$i \in \omega$$

(ННТ).

Нехай b також нерухома точка, тобто $b = \varphi(b)$. Маємо, що $1 \leq b$. За

$\varphi(\bigvee_{i \in \omega} d_i) = \bigvee_{i \in \omega} \varphi(d_i)$
 монотонністю $\varphi(\bigwedge_{i \in \omega} d_i) \leq \bigwedge_{i \in \omega} \varphi(d_i)$
 $\bigwedge_{i \in \omega} \varphi(d_i) \leq b$ для всіх $i \in \omega$. Тому $\varphi(\bigwedge_{i \in \omega} d_i) \leq b$.

$i \in \omega$

$(\bigwedge_{i \in \omega} d_i)$

Отже, $\varphi(\bigwedge_{i \in \omega} d_i)$ - найменша з нерухомих точок.

■

$i \in \omega$

Теорема 5.2 (структура множини нерухомих точок)

Нехай D - ω -область, φ - неперервне відображення на D .
 Тоді множина

нерухомих точок $Y = \{d \in D \mid \varphi(d) = d\}$ - ω -область.

Доведення

Доведемо, що виконуються три умови визначення ω -області.

1. Частковий порядок на множині нерухомих точок отримуємо як звуження

часткового порядку на D : $\leq_Y \subseteq \leq_D$

2. Найменшим елементом в Y є елемент $\text{lfp } \varphi$ (за теоремою Кнастера-Тарського-Кліні).

3. Доведемо повноту. Розглянемо ланцюг $d_0 \leq d_1 \leq d_2 \leq \dots$ елементів Y .

Покажемо, що границя ланцюга є нерухомою точкою, тобто $\bigvee d_i \in Y$. Дійсно,

i

$i \in \omega$

непер.

$\varphi(\bigvee_{i \in \omega} d_i) = \bigvee_{i \in \omega} \varphi(d_i) = \bigvee_{i \in \omega} d_i$. Тобто $\bigvee_{i \in \omega} d_i$ - нерухома точка.

$\bigwedge_{i \in \omega} \varphi(d_i) = \bigwedge_{i \in \omega} d_i$

■

$i \in \omega$ $i \in \omega$ $i \in \omega$ $i \in \omega$

Перш ніж рухатися далі у вивченні властивостей оператора найменшої нерухомої точки, доведемо важливу лему.

Лема 5.2 Якщо множина $\{d_i\}_{i \in \omega} \subseteq D$ впорядкована за обома індексами, тобто

$i < j \implies d_i \leq d_j$ $i, j \in \omega$

$\forall i \in \omega \forall j \in \omega (d_i \leq d_j \implies d_{i+1} \leq d_{j+1})$, то

$d_{i(j+1)} \leq d_{(i+1)j}$

$(\bigvee_{i \in \omega} d_i) = (\bigvee_{i \in \omega} d_{i+1}) =$

$\bigvee_{i \in \omega} d_i$.

C_{kk}
 C_{ij}
 C_{ij}
 $i \in \omega$
 $j \in \omega$
 $j \in \omega$
 $i \in \omega$
 k
 $\in \omega$

Доведення
 Попередньо необхідно показати, що
 множини, які розглядаються під
 супремумами, є ланцюгами (тобто $\{C_{dij} \mid j \in \omega\}$, $\{C_{dij} \mid j \in \omega\}$, $\{d_{kk} \mid k \in \omega\}$ - ланцюги). Це
 $j \in \omega$
 $i \in \omega$
 впливає з вибору множини $\{d\}$.
 $i, j \in \omega$

Метод доведення леми полягає в тому, що ми покажемо, що множини мажорант усіх ланцюгів співпадають. Звідси буде впливати рівність границь (бо вони є найменшими з мажорант).

Нехай a - мажоранта ланцюга $\{C_{dij} \mid j \in \omega\}$. Звідси a - мажоранта d_{ij} для всіх $i \in \omega$
 $j \in \omega$

та для всіх $j \in \omega$, а отже a - мажоранта для C_d для всіх $j \in \omega$, тобто a
 i, j
 $i \in \omega$

мажоранта для ланцюга $\{C_d \mid j \in \omega\}$. Аналогічно a - мажоранта для ланцюга $\{d \mid k \in \omega\}$.
 i, j
 k
 $i \in \omega$

Залишилося продемонструвати, що якщо a - мажоранта для $\{d_{kk} \mid k \in \omega\}$, то a -

мажоранта для $\{C_d \mid k \in \omega\}$. Дійсно, для будь якого елемента d існує k (зокрема,
 $i, j \in \omega$
 $j \in \omega$

$k = \max(i, j)$), що $d \leq d_{kk}$. Тому a - мажоранта для $\{C_d \mid i, j \in \omega\}$.
 $i, j \in \omega$

$j \in \omega$

5.5. Конструювання похідних ω -областей

У програмуванні поруч із базовими типами даних часто використовуються похідні (структуровані) дані. Для дослідження рівнянь над похідними даними доречно розглянути побудову похідних ω -областей з базових областей. Найпростішим методом побудови є об'єднання областей.

Розмічене об'єднання ω -областей

Не втрачаючи загальності, розглянемо випадок двох областей. Нехай D_1 та D_2

ω -області. Будемо індексувати відповідні поняття наступним чином: \leq_{D_1} , \perp_{D_1} , C_{D_1}

для першої області, \leq_{D_2} , \perp_{D_2} , C_{D_2} - для другої області. Нехай також D_1 та D_2 не

перетинаються. Будуємо нову множину $D = D_1 \cup D_2 \cup \{\perp\}$, де \perp - новий елемент, що не належить ні D_1 , ні D_2 .

На множині D вводимо частковий порядок наступним чином:

$\leq_D = \leq_{D_1} \cup \leq_{D_2} \cup \{(\perp, \perp)\} \cup \{(\perp, d_1) \mid d_1 \in D_1\} \cup \{(\perp, d_2) \mid d_2 \in D_2\}$

Інакше кажучи, новий частковий порядок є об'єднанням часткових порядків на D_1 та D_2 , і крім того елемент \perp визначено найменшим елементом в D .

Неважко довести, що колапсу так побудованого відношення часткового порядку не буде. Повнота D відносно порядку \leq_D впливає з його розміченості (множини D_1 та D_2 не перетинаються, тому ланцюг обов'язково складається з елементів однієї множини крім, можливо, \perp).

Лема 5.3. Розмічене об'єднання ω -областей є ω -областю.

Декартовий добуток ω -областей

Нехай D_1 та D_2 - ω -області. Побудуємо ω -область для $D = D_1 \times D_2$.

Задамо частковий порядок на D по координатно:

def $(d_1, d_2) \leq_D (d_1', d_2') \Leftrightarrow (d_1 \leq_{D_1} d_1') \ \& \ (d_2 \leq_{D_2} d_2')$.

Неважко довести, що так визначене відношення є частковим порядком.

Найменшим елементом D буде елемент $(\perp_{D_1}, \perp_{D_2})$.

Доведемо повноту D відносно введеного відношення. Нехай $\{(d_1, d_2)\}$ - ланцюг в

$i \in \omega$

D . З визначення часткового порядку на D випливає, що $\{d_1\}$ та $\{d_2\}$ - ланцюги

$i \in \omega$ $i \in \omega$

відповідно в D_1 та D_2 . З повноти цих областей випливає, що існують та

$$C_{D_1} d_1$$

$$D_1 i$$

$$i \in \omega$$

$C_{D_2} d_{2j}$. Очевидно, що пара $(C_{D_1} d_1, C_{D_2} d_{2j})$ є мажорантою елементів ланцюга $\{C_{D_1} d_1, C_{D_2} d_{2j}\}$. Можна показати, що ця мажоранта є найменшою. А це означає, що

$\{(d_1, d_2)\}$. Можна показати, що ця мажоранта є найменшою. А це означає, що

$$C_{D_1} (d_1, d_2) = (C_{D_1} d_1, C_{D_2} d_{2j}), \text{ тобто, що } D - \text{ повна.}$$

З наведеного доведення також випливають наступні співвідношення: $C_{D_1} d_1 \leq C_{D_2} d_{2j}$

$$d_1 \leq d_2$$

$$D_1 i$$

$$i \in \omega$$

$$= (C_{D_1} d_1, d_2) \text{ та } C_{D_1} (d_1, d_2) = (C_{D_1} d_1, C_{D_2} d_{2j}), \text{ де } d_1 \in D_1, d_2 \in D_2.$$

Отже, доведено наступне твердження.

Лема 5.4. Декартовий добуток ω -областей є ω -областю.

Лема 5.5. Нехай D_1 та D_2 - ω -області. Тоді

відображення $\varphi : D_1 \times D_2 \rightarrow R$ неперервне тоді і тільки тоді, коли воно неперервне за кожним аргументом.

Доведення

Необхідність. Нехай відображення $\varphi : D_1 \times D_2 \rightarrow R$ неперервне на множині $D = D_1 \times D_2$. Це означає, що для довільного ланцюга $(a, b) \leq (a, b) \leq (a, b) \leq \dots$

$$0 \leq C_{D_1} (a, b) \leq C_{D_2} (a, b) \leq C_{D_1} (a, b) \leq \dots$$

Щоб довести неперервність за першим аргументом розглянемо ланцюг $(a, b) \leq (a, b) \leq (a, b) \leq \dots$, у елементів якого другий компонент дорівнює b . Звідси $C_{D_1} (a, b) \leq C_{D_2} (a, b) \leq C_{D_1} (a, b) \leq \dots$

впливає, що $a_0 \leq a_1 \leq a_2 \leq \dots$ та

$$\varphi(a, b) = \left(\bigcap_{i \in \omega} \varphi(a_i, b_i) \right) = \left(\bigcap_{i \in \omega} \varphi(a_i, b_i) \right)$$

Використовуючи ці співвідношення, отримуємо, що

$$\varphi(a, b) = \left(\bigcap_{i \in \omega} \varphi(a_i, b_i) \right) = \left(\bigcap_{i \in \omega} \varphi(a_i, b_i) \right)$$

Аналогічно доводиться неперервність і за другим аргументом.

Достатність. Треба довести, що з неперервності за кожним аргументом випливає неперервність $\varphi : D_1 \times D_2 \rightarrow R$ в цілому. Беремо довільний ланцюг і користуючись властивостями супремуму на декартовому добутку та неперервністю за кожним аргументом, отримуємо наступні рівності:

$$\varphi(a, b) = \left(\bigcap_{i \in \omega} \varphi(a_i, b_i) \right) = \left(\bigcap_{i \in \omega} \varphi(a_i, b_i) \right)$$

Класи функцій на ω -областях. Доведемо, що множина тотальних (всюди визначених) функцій $F = [D \rightarrow D]$ на ω -області є ω -областю.

Щоб відрізнити елементи та відношення на множинах F та D будемо вживати відповідні індекси, позначаючи частковий порядок на D як \leq_D , а на F - як \leq_F .

Спочатку визначимо частковий порядок \leq_F на $F = [D \rightarrow D]$ наступним чином.

Нехай $f, g \in F$. Тоді вважаємо, що

$$f \leq_F g \Leftrightarrow \forall d \in D (f(d) \leq_D g(d))$$

Для так введеного бінарного відношення виконуються всі аксіоми часткового порядку: рефлексивність, транзитивність, антисиметричність.

Дійсно, оскільки $\forall d \in D (f(d) \leq_D f(d))$, то $f \leq_F f$ (рефлексивність). Далі,

$\forall d \in D (f(d) \leq_D g(d) \wedge g(d) \leq_D h(d) \Rightarrow f(d) \leq_D h(d))$,
тому

$$f \leq_F g \wedge g \leq_F h \Rightarrow f \leq_F h$$

$f \leq_F g, g \leq_F h \Rightarrow f \leq_F h$ (транзитивність). Нарешті,
справедливе співвідношення

$\forall d \in D ((f \leq_D g(d), (g \leq_D f) \Rightarrow (f \leq_D g)))$,
з якого
впливає $f \leq_F g, g \leq_F f \Rightarrow f =_F g$ (антисиметричність).

Тепер визначимо найменший елемент $\perp_F =$
 $[d \in D \mid d \in D]$. Тут вираз \perp
 квадратних дужках називається функціональним
 конструктором. Він задає

def
 значення функції на кожному її аргументі. Це означає, що $\forall d \in D (\perp ($
 $)d = \perp)$.

$F \subseteq D$
 Тому для довільної функції $g \in F$ маємо, що $\perp ($
 $)d = \perp \leq (g)d$. За визначенням, це

$D \subseteq D$
 означає, що $\perp_F \leq_F g$.

Далі треба продемонструвати повноту F . Нехай f_0
 $\leq_F f_1 \leq_F f_2 \leq K$ – ланцюг в F .

Треба довести, що $\sup_{i \in \omega} f_i \in F$, тобто, що супремум функцій є
 функцією.

$$\sup_{i \in \omega} f_i$$

Спочатку визначимо нову функцію $f = [d$
 $\in C (f_i \mid d \in D)$. Наведений

$D \subseteq$

$i \in \omega$

функціональний конструктор задає значення функції на кожному її
 аргументі. Це

def

означає, що $\forall d \in D (f(d) = \sup_{i \in \omega} (f_i(d)))$.

Доведемо, що $\sup_{i \in \omega} f_i = f$ (тобто що f –
 найменша з мажорант $\{ f_i \mid i \in \omega \}$ ланцюга

$f_0 \leq_F f_1 \leq_F f_2 \leq K$).

1. Спочатку доведемо, що f - мажоранта множини функцій $\{ f_i \}_{i \in \omega}$. Дійсно, за

def

визначенням $f \quad \forall d \in D \quad ((f)_d = C (f_i)_d)$.

Звідси випливає, що $\forall d \in D$

$$D \quad i \\ i \in \omega$$

$\forall i \in \omega \quad ((f_i)_d \leq_D (f)_d)$. Переставляючи універсальні квантори, отримуємо, що

$$\forall i \in \omega \quad \forall d \in D \quad (f_i)_d \leq_D (f)_d$$

А це означає (за визначенням часткового порядку на F), що $\forall i \in \omega \quad f_i \leq_F f$. Тобто,

f - мажоранта множини функцій $\{ f_i \}_{i \in \omega}$.

2. Доведемо тепер, що f - найменша мажоранта множини функцій $\{ f_i \}_{i \in \omega}$.

Нехай g - довільна мажоранта цієї множини. Тоді

$$\forall i \in \omega \quad f_i \leq_F g \Leftrightarrow \forall i \in \omega \quad \forall d \in D \quad (f_i)_d \leq g(d) \Leftrightarrow \forall d \in D$$

$$\forall i \in \omega \quad (f_i)_d \leq g(d) \Leftrightarrow \forall d \in D$$

$$i \quad D$$

$$C (f_i)_d \leq g(d) \Leftrightarrow \forall d \in D \quad (f)_d \leq_D g(d) \Leftrightarrow f \leq_F g$$

$$D \quad i \\ i \in \omega$$

Це означає, що доведена наступна лема.

def

Лема 5.6. Відображення $f = [d \mapsto C (f_i)_d \mid d \in D]$ є супремумом множини

$$D \quad i \\ i \in \omega$$

функцій $\{ f_i \}_{i \in \omega}$, тобто $f = C f_i$.

З доведених властивостей випливає, що множини неперервних функцій $\epsilon \omega$ -областю. Отриманий результат можна посилити для випадку множини функцій

$[D_1 \rightarrow D_2]$ на ω -областях D_1 та D_2 .

Лема 5.7. Множина функцій, визначених на ω -областях, є ω -областю.

довести, що f - неперервна, треба довести, що $f(C d) = C f(d)$ для довільного

$$D_i \quad D_i$$

$$i \in \omega \quad i \in \omega$$

ланцюга $\{d_i\}$. Маємо, що $(C f)(C d) = C(f(C d))$.

$$i \in \omega \quad F_j \quad D_i \quad D_j \quad D$$

$$i \quad D \quad D_j \quad i \quad j \in \omega \quad i \in \omega \quad j \in \omega \quad i$$

$$\in \omega \quad j \in \omega \quad i \in \omega$$

За лемою 5.2 $C(C f)(d) = C(C f(d))$, тому

$$C(C f)(d) = C(C f(d))$$

$$D_j \quad i \quad D_j \quad i$$

$$j \in \omega \quad i \in \omega \quad i \in \omega \quad j \in \omega$$

$$C(C f)(d) = C((C f)(d)) = C f(d)$$

$$D_j \quad i \quad D \quad i \quad i \in \omega \quad j \in \omega \quad i \in \omega$$

с

Це означає, що клас $[D \rightarrow D]$ є ω -областю. Отриманий результат можна посилити

с

для випадку множини неперервних функцій $[D_1 \rightarrow D_2]$, заданих на ω -областях D_1 та

D_2 . Це дозволяє сформулювати наступний результат.

Лема 5.9. Множина неперервних функцій, заданих на ω -областях, є ω -областю.

5.6 Властивості оператора найменшої нерухомої точки

Операцію взяття нерухомої точки можна трактувати як оператор

с с

$lfp: [D \rightarrow D] \rightarrow D$, де $[D \rightarrow D]$ - множина неперервних відображень.

Доведемо, що оператор lfp - неперервний. Але спочатку покажемо, що він - монотонний.

с

Лема 5.10. Відображення $lfp: [D \rightarrow D] \rightarrow D$ - монотонне відображення, тобто

с

$$g \leq_F h \Rightarrow lfp(g) \leq_D lfp(h) \text{ для } g, h \in [D \rightarrow D].$$

Доведення. Нехай $g \leq_F h$. Розглянемо наступні співвідношення:

$$\perp_D \leq \perp_D$$

(Умова $g \leq_F h$)

$$g(\perp_D) \leq h(\perp_D)$$

(Монотонність g)

$$g(g(\perp_D)) \leq g(h(\perp_D)) \leq h(h(\perp_D))$$

та умова $g \leq_F h$)

...

...

...

(Монотонність g)

$$g(k)(\perp_D) \leq \dots \leq g(k-m)(h(m)(\perp_D)) \dots \leq h(k)(\perp_D)$$

та умова $g \leq_F h$)

...

...

...

$$\bigcup_{i \in \omega} g(\perp_D)^i \leq \bigcup_{i \in \omega} h(\perp_D)^i$$

Оскільки $\text{lfp}(g) = \bigcup_{i \in \omega} g(\perp_D)^i$, то $\text{lfp}(g) \leq \text{lfp}(h)$.

D

■

$i \in \omega$

$i \in \omega$

Теорема 5.3 (неперервність lfp). Якщо D - ω -область, то відображення lfp -

$c \quad c$

неперервне, тобто $\text{lfp} \in [D \rightarrow D] \rightarrow D$.

Доведення. Неперервність lfp означає, що для довільного ланцюга $f_0 \leq_F f_1 \leq_F f_2 \leq \dots$ має виконуватись співвідношення

$$\text{lfp}(\bigcup_{i \in \omega} f_i) = \bigcup_{i \in \omega} \text{lfp}(f_i)$$

Спочатку доведемо, що $\text{lfp}(C f) \geq C \text{lfp}(f)$.
 Оскільки $C f \geq f$, то за лемою $F i \leq D i$ для $i \in \omega$.

5.10 $\text{lfp}(C f) \geq \text{lfp}(f)$ для довільного $i \in \omega$, а отже $\text{lfp}(C f) \geq C (\text{lfp}(f))$.
 Для довільного $i \in \omega$, маємо $F i \leq D i$ для $i \in \omega$.

Тепер доведемо, що $\text{lfp}(C f) \leq C (\text{lfp}(f))$. Для цього покажемо, що $a = C \text{lfp}(f)$ є нерухомою точкою для f .
 Дійсно, $f(a) = (C f)(C \text{lfp}(f)) = C(f(C \text{lfp}(f))) = C(C(f(\text{lfp}(f))))$.
 Позначимо $d_{ij} = f i (\text{lfp}(f j))$, $i, j \in \omega$.

Неважко довести, що засновки леми 5.2 виконуються, тому $d_{ij} = f i (\text{lfp}(f j))$, $i, j \in \omega$.
 Дійсно, $f(a) = (C f)(C \text{lfp}(f)) = C(f(C \text{lfp}(f))) = C(C(f(\text{lfp}(f))))$.
 Позначимо $d_{ij} = f i (\text{lfp}(f j))$, $i, j \in \omega$.

Неважко довести, що засновки леми 5.2 виконуються, тому $d_{ij} = f i (\text{lfp}(f j))$, $i, j \in \omega$.
 Дійсно, $f(a) = (C f)(C \text{lfp}(f)) = C(f(C \text{lfp}(f))) = C(C(f(\text{lfp}(f))))$.
 Позначимо $d_{ij} = f i (\text{lfp}(f j))$, $i, j \in \omega$.

Неважко довести, що засновки леми 5.2 виконуються, тому $d_{ij} = f i (\text{lfp}(f j))$, $i, j \in \omega$.

$$C_D (C_D (f_i (\text{lfp}(f_j)))) = C_k f_k (\text{lfp}(f_k)) = C_D (\text{lfp}(f)) = a.$$

Оскільки $\text{lfp}(C f)$ - найменша нерухома точка, то вона менша за будь-яку нерухому точку для $C f$.

Тому $\text{lfp}(C f) \leq C \text{lfp}(f) = a$.
 Отже ми довели, що $\text{lfp}(C f) = \text{lfp}(f)$.

Отже ми довели, що $\text{lfp}(C f) = \text{lfp}(f)$.
 ■

Таким чином, досліджено основні властивості оператора найменшої нерухомої точки, який може вважатися одним із уточнень рекурсії.

5.7. Застосування теорії ННТ

Найважливішими способами визначення нескінченних об'єктів (множин, послідовностей, функцій і таке інше), є рекурентні, індуктивні та рекурсивні способи. Оскільки перші два способи є частковими випадками рекурсії, то звідси випливає важливість теорії найменшої нерухомої точки (ННТ), як одного із способів уточнення рекурсії, в різних застосуваннях. Тут ми розглянемо застосування ННТ для уточнення синтаксису та семантики мов програмування.

5.7.1. Уточнення синтаксису мов програмування

Синтаксис мов програмування переважно задається за допомогою БНФ, граматик та інших подібних формалізмів. Сама структура правил БНФ говорить про можливість застосування теорії ННТ для уточнення формальних мов, що задаються БНФ.

У попередньому розділі було розглянуто приклад подання породжуючої граматики як системи рівнянь.

Так, правила $A \rightarrow aAb \mid \varepsilon$ задають мову, яка є розв'язком рівняння

$A = \{a\}A\{b\}U\{\varepsilon\} = \varphi(A)$. Але для того, щоб стверджувати існування розв'язку, необхідно дослідити область, на якій задаються такі рівняння.

Розглянемо слабку алгебру формальних мов, в якій задано наведене рівняння

(відображення φ):

$$A_L = \langle 2, \overset{*}{T}, U \cdot \rangle, \text{ де}$$

$2T$ – множина мов, U та \cdot – операції об'єднання та конкатенації.

Достатніми умовами існування розв'язків рекурсивних рівнянь в алгебрах

наведеного типу є умови для множини $2T$ бути ω -областю, та для операцій

об'єднання та конкатенації бути неперервними. Доведемо виконання цих умов.

Лема 5.11. Множина $2T$ – ω -область

Доведення. Щоб множина була ω -областю треба визначити на ній частковий порядок та показати його повноту та вказати найменший елемент.

*

Для множини $2T$ частковим порядком буде включення, порожня множина

найменшим елементом, а об'єднання множин буде супремумом, тобто є така

відповідність:

- 1) \leq - \subseteq
- 2) \perp - $\{\}$
- 3) CL_i - UL_i
 $i \in \omega$ $i \in \omega$

Очевидно, що всі необхідні умови ω -області виконуються для так введеного часткового порядку. ■

Лема 5.12. Операції об'єднання та конкатенації - ω -неперервні.

Доведення. Розглянемо операцію об'єднання U . Згідно леми 5.5 досить довести неперервність об'єднання за кожним аргументом. Не обмежуючи загальності, доведемо неперервність за першим аргументом.

*

*

Нехай $\{L_i\}_{i \in \omega}$ - ланцюг з множини $2T$, $L' \in 2T$, $L = \bigcup_{i \in \omega} L_i$. За визначенням,

$i \in \omega$

$CL_i = UL_i$. Щоб довести неперервність, треба показати, що $(\bigcup_{i \in \omega} UL_i) \cup L' = U(\bigcup_{i \in \omega} L_i \cup L')$.

Дійсно, для довільного елемента x маємо:

$$x \in (\bigcup_{i \in \omega} UL_i) \cup L' \Leftrightarrow x \in UL_i \vee x \in L' \Leftrightarrow \exists k \exists x (x \in L_k \vee x \in L') \Leftrightarrow \exists k \exists x (x \in L_k \cup L') \Leftrightarrow x \in U(\bigcup_{i \in \omega} L_i \cup L')$$

$U(\bigcup_{i \in \omega} L_i \cup L')$.

$i \in \omega$

Отже, U - неперервна операція.

Розглянемо тепер операцію \cdot . За визначенням,

$$L \cdot L = \{ l_1 l_2 \mid l_1 \in L, l_2 \in L \}.$$

Доведемо, що $(\bigcup_{i \in \omega} L_i) \cdot L = \bigcup_{i \in \omega} (L_i \cdot L)$. Маємо, що

$$x \in (\bigcup_{i \in \omega} L_i) \cdot L \Leftrightarrow \exists z \exists y (y \in (\bigcup_{i \in \omega} L_i), z \in L, x = yz) \Leftrightarrow$$

$$\exists i \in \omega$$

$$yz \in L_i \cdot L, x = yz \Leftrightarrow \exists k \exists y (y \in L_i, z \in L, x = yz) \Leftrightarrow \exists k \exists y (y \in L_i \cdot L, x = yz) \Leftrightarrow$$

$$x \in \bigcup_{i \in \omega} (L_i \cdot L).$$

Отже, операція \cdot також неперервна. ■

Таким чином, множина формальних мов над певним алфавітом є ω -областю, а операції слабкої алгебри формальних мов - неперервними. Похідні операції також будуть неперервними. Тому рекурсивні визначення (рівняння) над цією алгеброю завжди мають найменший розв'язок. Цей розв'язок, за теоремою КТК, можна знайти методом послідовних наближень.

Доведемо, що якщо за KB-граматикою побудувати відповідну систему рекурсивних визначень, то розв'язок цієї системи співпаде з мовою, породженою граматикою.

Отже, нехай $G = (N, T, P, S)$ - KB-граматика, $\alpha \in (N \cup T)^*$. Задамо відображення

$$T$$

$s: (N \cup T) \rightarrow (N \cup T^2)$, вважаючи, що $s(A) = A$, $A \in N$, та $s(a) = \{ a \}$, $a \in T$. Це відображення

розповсюджуємо на послідовності символів з алфавіту (Σ^*) , вважаючи, що

T

$s(x_1 x_2 \dots x_n) = s(x_1) s(x_2) \dots s(x_n)$. Тепер маємо, що $s: (\Sigma^*)^* \rightarrow (\Sigma^2)^*$. Сукупність усіх

правил $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \dots, A \rightarrow \alpha_k$ з граматики G з одним і тим же нетерміналом у лівій

частині замінюємо на рекурсивне визначення $A = s(\alpha_1) \cup s(\alpha_2) \cup \dots \cup s(\alpha_k)$. Такі

визначення будемо для всіх нетерміналів. Отриману систему рекурсивних визначень над слабкою алгеброю формальних мов позначимо $E(G)$. Відображення, що задається правими частинами цієї системи позначимо Φ , його складову за нетерміналом A позначимо Φ_A , а найменшу нерухому точку цього відображення за

змінною (нетерміналом) S позначимо $\text{lfp}_{S \in G}(\Phi)$.

Приклад 5.1. Для граматики $G = (\{S, A, B, C, D\}, \{a, b, c\}, P, S)$, де P складається з наступних правил:

- $S \rightarrow AB \mid CD \mid \varepsilon$
- $A \rightarrow \varepsilon \mid aA$
- $B \rightarrow \varepsilon \mid bBc$
- $C \rightarrow \varepsilon \mid aCb$
- $D \rightarrow \varepsilon \mid cD$

отримуємо наступну систему рекурсивних визначень:

- $S = AB \cup CD \cup \{\varepsilon\}$
- $A = \{\varepsilon\} \cup \{a\}A$
- $B = \{\varepsilon\} \cup \{b\}B\{c\}$
- $C = \{\varepsilon\} \cup \{a\}C\{b\}$
- $D = \{\varepsilon\} \cup \{c\}D$.

Доведемо тепер, що мова, породжена КВ-граматикою, співпадає з найменшим розв'язком системи рекурсивних визначень (рівнянь), побудованих за цією граматикою.

Теорема 5.4. Для КВ-граматики $G = (N, T, P, S)$

$$L(G) = \text{lfp}_{S \in G}(\Phi).$$

Доведення. Нехай $L(k)A(G)$ - множина термінальних ланцюжків породжених з нетерміналу A деревами виводу глибини не більше k . Індукцією за k доводимо, що

$$L(k)A(G) = \Phi(k)A(\perp).$$

Дійсно, для виводів довжини 0 маємо, що $L(0)A(G) = \emptyset$ та $\Phi(0)A(\perp) = \perp = \emptyset$.

Припускаючи, що індуктивна гіпотеза вірна для k , доведемо її справедливості для $k+1$. Дійсно, мова $L(k+1)A(G)$ складається з термінальних ланцюжків t , які виводяться з A деревами виводу глибини не більше $k+1$. Такі дерева породжуються деяким правилом $A \rightarrow x_1 x_2 \dots x_n$, $x_i \in P$, а виводи з нетермінальних символів x_1, x_2, \dots, x_n мають глибину не більшу k . Термінальні ланцюжки, що породжуються з таких нетерміналів x_i входять у мову $\Phi(k)A(\perp)$. З побудови $\Phi(k+1)A(\perp)$ випливає, що ланцюжок t буде належати цій мові. Також є вірним і зворотне: якщо $t \in \Phi(k+1)A(\perp)$, то $t \in L(k+1)A(G)$.

Індуктивна гіпотеза доведена. Звідси випливає справедливості теореми. ■
 Таким чином, для подання формальних мов можна використовувати формалізм рекурсивних визначень та теорію ННТ. Правда, цей формалізм є більш потужним, якщо допускається використання нескінченних мов у визначеннях. Разом з тим неважко довести, що для слабо рекурсивних визначень (із скінченними мовами у визначеннях), найменшими розв'язками будуть лише КВ-мови.

Вибір формалізму подання синтаксису залежить від способу його використання (від прагматики).
 Тепер розглянемо використання теорії ННТ для подання семантики мов програмування.

5.7.2. Семантика мов програмування

У першому розділі посібника семантика програм мови SIPL задавалась функціями програмної алгебри

$$A_Prog = \langle FNA, FNB, FNAB, FA, FB, FS; S, AS, \cdot, IF, WN, x \Rightarrow, id \rangle.$$

Програми мови SIPL не є рекурсивними, але легко зробити таке її розширення, яке буде мати рекурсивні функції та процедури (це буде зроблено трохи пізніше). Тому

здається вірогідним, що класи часткових функцій алгебри A_Prog (класи FA, FB, FS) є ω -областями, а композиції, задані на цих класах, – неперервними. Це, дійсно, так.

Доведемо більш загальний результат. Нехай D, R – деякі множини даних, а $F = D \rightarrow R$ – клас часткових функцій, заданих на цих множинах. Введемо на F

відношення часткового порядку \leq , вважаючи, що для функцій $f, g \in F$

$$f \leq g \Leftrightarrow \forall d (d \in D_f \Rightarrow f(d) \downarrow \wedge f(d) \downarrow \Rightarrow g(d) \downarrow)$$

де $f(d) \downarrow$ означає, що функція f визначена на d .

Якщо використовувати графіки функцій, то можна записати, що

$$f \leq g \Leftrightarrow (gr) f \subseteq gr()g .$$

Очевидно, що так введене бінарне відношення ϵ , дійсно, відношенням частково порядку. Найменшим елементом буде всюди невизначена функція, для якої

залишимо позначення \perp .

Супремумом ланцюга функцій $f_0 \leq f_1 \leq f_2 \leq K$ буде функція $f = \bigcup_{i \in \omega} f_i$, графік якої є

$$i \in \omega$$

об'єднання графіків функцій ланцюга, тобто $(gr) f = \bigcup_{i \in \omega} Ugr(f_i)$. Неважко переконатися,

$$(\langle a, b \rangle, \langle c, a \rangle) \in f .$$

це об'єднання є функціональним бінарним відношенням. Дійсно, нехай Це означає, що

$$(\langle a, b \rangle, \langle c, a \rangle) \in Ugr(f_i) \Leftrightarrow (\exists k, \langle a, b \rangle) \in gr(f_k) ,$$

$$i \in \omega$$

$$(\langle a, c \rangle, \langle m, a \rangle) \in Ugr(f_i) \Leftrightarrow (\exists m, \langle a, c \rangle) \in gr(f_m) .$$

$$i \in \omega$$

$$(\langle a, b \rangle, \langle c, a \rangle) \in gr(f_k) \square$$

Якщо $k \geq m \Rightarrow \square \Rightarrow b = c$, аналогічно розглядається випадок $k \leq m$.

$$(\langle a, c \rangle, \langle m, a \rangle) \in gr(f_k) \square$$

Тобто $f \in \epsilon$, дійсно, функцією. Це говорить про повноту F .

Таким чином, доведено наступне твердження.

Лема 5.13. Множина $F = D \rightarrow R$ часткових функцій із D в R є ω -область відносно введеного часткового порядку.

Зауважимо відмінність цієї леми від лем 5.7 та 5.9, які говорять про класи тотальних функцій, частковий порядок на яких індукується частковим порядком на їх областях визначення та значень. У лемі 5.3 говориться про клас часткових функцій, визначених на довільних множинах, які не обов'язково є ω -областями. В зв'язку з цим класи n -арних тотальних функцій $FNA, FNB, FNAB$ не підпадають під дію леми 5.13.

Застосовуючи лему 5.13 до наступних класів (основ алгебри) часткових функцій алгебри $A_Prog: FA, FB, FS$, отримуємо, що ці основи є ω -областями.

$$n \quad x$$

Доведемо тепер, що композиції $S, AS, \cdot, IF, WH, x \Rightarrow, id$ алгебри A_Prog є неперервними.

Це очевидно для нуль-арних композицій (функцій) $x \Rightarrow id$, які не мають функцій-

аргументів.

Лема 5.14. Композиція суперпозиції S_n номінативних функцій у n -арну функцію неперервна за номінативними функціями-аргументами.

Доведення. Нехай f - n -арна функція, g_1, \dots, g_n - номінативні функції типу FA ,

$$f: A^1 \rightarrow A^n$$

$f_0 \leq f_1 \leq f_2 \leq \dots \leq K$ - ланцюг функцій з FA , $st \in State$. Нагадаємо, що:

$$S_n(f, g_1, \dots, g_n)(st) = f(g_1(st), \dots, g_n(st)).$$

Доведемо неперервність суперпозиції за першим аргументом типу FA , тобто

$$\text{доведемо, що } S_n(f, f_i, \dots, g_n) = S_n(f, f_i, \dots, g_n).$$

Нехай $S_n(f, f_i, \dots, g_n)(st) \downarrow = r$. Отримуємо наступну послідовність еквівалентних

$$C_i \quad n$$

перетворень:

$$\begin{aligned} S_n(f, f_i, \dots, g_n)(st) \downarrow = r &\Leftrightarrow f(g_1(st), \dots, g_n(st)) \downarrow = r \\ &\Leftrightarrow \exists k (f(g_1(st), \dots, g_n(st)) \downarrow = r \Leftrightarrow S_n(f, f_i, \dots, g_n)(st) \downarrow = r). \end{aligned}$$

Наведені перетворення спираються на той факт, що

$$f(g_1(st), \dots, g_n(st)) \downarrow = r' \Leftrightarrow \exists k (f(g_1(st), \dots, g_n(st)) \downarrow = r' \Leftrightarrow S_n(f, f_i, \dots, g_n)(st) \downarrow = r').$$

Аналогічним чином доводиться наступне твердження.

Лема 5.15. Композиція присвоєння AS - неперервна.

Доведемо неперервність послідовного виконання.

Лема 5.16. Композиція послідовного виконання \cdot - неперервна.

Доведення. Щоб довести неперервність композиції \cdot , нам потрібно довести

неперервність • за кожним аргументом. Доведемо лише неперервність за першим

аргументом, тобто, що $(Cf i) \cdot g = F C(f i \cdot g)$ для довільного ланцюга $f_0 \leq f_1 \leq f_2 \leq K$

функцій з FS. Маємо, що

$$\begin{aligned} & ((Cf i) \cdot g)(st) \downarrow = r \Leftrightarrow \exists b (Cf i)(st) \downarrow = b \ \& \ g(b) \downarrow = r \Leftrightarrow \\ & \Leftrightarrow \exists b k \exists f (k)st \downarrow = b \ \& \ g(b) \downarrow = r \Leftrightarrow \exists k (f k \cdot g)(st) \downarrow = r \Leftrightarrow \\ & \Leftrightarrow (C i (f \cdot g))(st) \downarrow = r \end{aligned}$$

Лема 5.17. Композиція IF – неперервна.

Доведення. Щоб довести неперервність композиції IF, нам потрібно довести

неперервність IF за кожним аргументом. Розглянемо лише доведення неперервності

за одним аргументом, оскільки за іншими вона доводиться аналогічно. Нагадаємо, що

$$\begin{aligned} & \square f(st), \text{ якщо } p(st) \downarrow = \text{true} \\ & \square (p, IF, f)(g st) = g(st), \text{ якщо } p(st) \downarrow = \text{false} \\ & \square \text{невизначено в інших випадках} \\ & \square \end{aligned}$$

Розглянемо перший випадок: $p(st) \downarrow = \text{true}$. Тоді

$$\begin{aligned} & IF(p, f, g)(st) \downarrow = r \Leftrightarrow (C i (f \cdot g))(st) \downarrow = r \\ & \Leftrightarrow \exists k \exists f (k)st \downarrow = r \Leftrightarrow \exists k \exists f (f k \cdot g)(st) \downarrow = r \Leftrightarrow \\ & \Leftrightarrow \exists k \exists f (C i (f \cdot g))(st) \downarrow = r \end{aligned}$$

Другий випадок $p(st) \downarrow = \text{false}$ є більш простим.

Для доведення неперервності композицій ітерації (циклу) спочатку покажемо, що

ітерація функцій є найменшою нерухомою точкою певного рекурсивного визначення.

Теорема 5.5. Для композиції циклу виконується наступне співвідношення:

$$(WH, p) f = \text{lfp} X \text{ IF}(p, f X \cdot \text{id}$$

,)

Доведення

Раніше (лема 1.2) було доведено, що $(WH, p) f$ є нерухомою точкою рівняння

$X = \text{IF}(fb, fs \cdot X, \text{id})$, звідки випливає, що $(WH, p) f \geq \text{lfp} X \text{ IF}(p, f X \cdot \text{id},)$. Тому залишилось довести, що $(WH, p) f \leq \text{lfp} X \text{ IF}(p, f X \cdot \text{id},)$.

Попередньо нагадаємо початкове визначення композиції циклу WH (розділ 1):

$WH(fb, fs)(st) = st$, де $st = st$, $st = fs(st)$, $st = fs(st)$, ..., $st = fs(st)$, причому $fb(st) = \text{true}$,

$fb(st) = \text{true}, \dots, fb(st) = \text{true}, fb(st) = \text{false}$.

Тепер позначимо $\text{IF}(p, f X \cdot \text{id},) = \Phi(X)$. Тоді $()_i$. Також

lfp

$$\text{IF}(p, f X \cdot \text{id},) = \text{C}\Phi() \perp$$

x

$i \in \omega$

будемо позначати $(WH, p)(f st) \downarrow \leq k = st_k$ той факт, що обчислення $WH(, p f)$ на даному st

завершилося з результатом st_k при числі ітерацій, не більше ніж k , тобто $\text{NumIt}_{WH}(p, f, st) \leq k$.

$\leq k$

$(-1)k +$

Доведемо індукцією за k , що з умови $(WH, p)(f st) \leq k$, випливає $(\Phi(1))(st) \downarrow = st$.

\downarrow

k (Усі твердження такого типу вважаються універсально квантифікованими, в даному випадку за st та st_k .)

База індукції ($k = 0$). З умови $(WH, p)(f st) \leq 0 = st$, випливає, що $st = st$, та

\downarrow

k 0

(1)

$p(st) = \text{false}$. Оскільки $\Phi(1) \perp = (\Phi) \perp = \text{IF}(p, f \cdot \perp, \text{id})$, то за вказаних умов будемо мати, що (1)

$$\Phi(1)(st) \downarrow = st.$$

Індуктивний крок. З припущення, що індуктивна гіпотеза справедлива для k ,

доведемо її справедливість для $k+1$.

Дійсно, нехай $(WH, p)(f st) \leq k+1 = st$. Тоді з леми 1.2 випливає, що

$(WH, p)(f f(st)) \downarrow \leq k+1 = st_{k+1}$ та що $p(st) = \text{true}$. За індуктивною гіпотезою маємо, що

$((-1)k + (-2)k +$

$(-2)k +$

$\varphi \quad (\perp) \quad (f \text{ (st)}) \downarrow = \text{stk} + 1.$ Звідси випливає, що $(\varphi$
 $(\perp) \quad (st) \downarrow = \text{stk} + 1,$ бо $\varphi \quad (\perp) =$
 $(\perp)^{k+1}$
 $\text{IF}(p, f \cdot \varphi \quad (\perp), \perp \text{ id})$ і при обчисленні маємо, що
 $(\perp)^{k+1}$
 $(\varphi \quad (\perp) \quad (st)) = (\text{IF}(p, f \cdot \varphi \quad (\perp)$
 $(\perp, \perp \text{ id}) \quad (st)) =$
 $(\perp)^{k+1}$
 $= f \cdot \varphi \quad (\perp) \quad (st) = (\varphi \quad (\perp) \quad (f$
 $(st)) = st \quad .$

$k+1$ ■

Лема 5.18. Композиція WH - неперервна.

Доведення. Скориставшись доведеними
 результатами щодо циклу та послідовного виконання
 неперервності композицій оператора, та умовного виконання
 отримуємо, що

$$\text{IF}(\perp, p \text{ C}(f \cdot X), \perp \text{ id}) = \text{WH}(\perp, p \text{ C}(f \cdot X)) = \text{lfp}_i (\text{IF}, (p \text{ C}(f \cdot X) \cdot \text{id}, \perp)) = \text{lfp}_{i \in \omega} (\text{IF}, (p \text{ C}(f \cdot X) \cdot \text{id}, \perp)) = \text{lfp}_i \text{CIF}(p, f \cdot X \cdot \text{id}, \perp) = \text{Clfp}_i \text{IF}(p, f \cdot X \cdot \text{id}, \perp) = \text{CWH}(p, f \cdot X \cdot \text{id}, \perp).$$

Аналогічним чином доводиться неперервність i за першим аргументом. ■

Доведені леми про неперервність композицій алгебри A_Prog дозволяють

стверджувати, що похідні композиції, які можна визначити в цій алгебрі, також

будуть неперервними. (Питання про точне визначення похідних композицій буде розглянуто пізніше.)

Факт неперервності композицій алгебри A_Prog не є випадковим. Має місце наступне твердження.

Твердження 5.1. Програмні композиції (які обчислюються за скінченну кількість кроків) - неперервні.

Доведення ґрунтується на тому факті, що для обчислення значень функцій,

побудованих за допомогою програмних композицій, використовується лише

скінченна частина (скінченний графік) функцій-аргументів. Деталі доведення тут не наводимо.

Таким чином, найважливіший висновок з теорії ННТ полягає в тому, що семантика мов програмування задається класами функцій, які є ω -областями, а композиції таких функцій є неперервними операторами. Це дозволяє обґрунтувати коректність використання різних рекурсивних методів для визначення засобів конструювання програм, спираючись на теореми, доведені в теорії ННТ.

Проілюструємо введення рекурсивних визначень функцій для розширення мови SIPL.

5.7.3. Рекурсивні розширення мови SIPL

Спочатку розглянемо приклади рекурсивних визначень функцій.

Приклад 5.1. Функцію $n!$ часто задають наступним рекурсивним визначенням:

$$f(n) = \text{if } n > 1 \text{ then } n * f(n-1)$$

else 1.

Головна складність таких визначень полягає у відсутності їх стандартного

семантичного уточнення. Одне з можливих уточнень має операційний характер, а

саме, вважається, що при обчисленні значення, наприклад, $f(5)$, обчислюється права

частина рекурсивного визначення (відбувається розгортка визначення при заміні n

на 5). Це обчислення призведе до необхідності обчислення $f(4)$, що у свою чергу потребує обчислення $f(3)$, і т.ін. І тільки обчислення $f(1)$ дасть результат 1, що дозволить обчислити $f(2)$, $f(3)$, $f(4)$, $f(5)$.

Аналіз цієї процедури обчислень показує, що обчислення фактично пов'язані з перетворенням виразів, їх «переписуванням», «розгорткою» для нових аргументів

функцій (англійською – rewriting). Але порядок застосувань переписуючих правил

також чітко не визначений. Тому виникають різні можливості розгортки рекурсивної функції, коли замінюється праве входження функції, чи ліве, чи внутрішнє, чи

зовнішнє. Цей порядок має визначатись семантикою визначень, тому ми знову

повертаємось до питання про семантику рекурсивних визначень.

Тут будемо застосовувати композиційно-номінативну семантику виразів. Це

означає, що права частина буде тлумачитись як номінативна функція, побудована з простих функцій за допомогою композицій.

Перше питання, що виникає, є наступним: яка номінативна функція відповідає виразу $\text{if } n > 1 \text{ then } n * f(n-1) \text{ else } 1$? Попереднє тлумачення, коли ми вживали

виклики функції виду $f(5)$, $f(4)$ і т.ін., говорить про те, що f є унарною функцією типу

$\text{Int} \rightarrow \text{Int}$. Тут фактично n є позначенням числа. Таке тлумачення для програмування не є вдалим, тому що для програмування є природним вживання функцій над

іменованими значеннями (детальніше це буде обговорюватись у подальших

розділах). Тому будемо вважати, що виразу $\text{if } n > 1 \text{ then } n * f(n-1) \text{ else } 1$ відповідає

функція, задана на станах змінних, причому n є однією із змінних. Але оскільки ми

вже «зарезервували» n для позначення довільного числа, будемо змінну позначати

прописною (великою) літерою N . Це означає, що нам краще переписати визначення саме у такій формі:

$$f(N) = \text{if } N > 1 \text{ then } N * f(N-1) \text{ else } 1$$

Отже, права частина визначення задає функцію, що обчислюється на даних виду

[Nan] та на їх розширеннях.

При побудові семантичного терму цього виразу виникає питання про тлумачення

2

виразу $f(N-1)$. Зрозуміло, що це буде суперпозиція функції $S(\text{sub}, N \Rightarrow, 1)$, яка

задається виразом $N-1$, в функцію f . Але функція f є номінативною, а не n -арною

функцію. Тому треба визначити суперпозицію у номінативні функції. Таких

суперпозицій може бути декілька, зокрема повна (глобальна) та неповна (локальна) суперпозиції. Для глобальної суперпозиції аргумент функції f формується повністю

(як цілісність). Для локальної суперпозиції формуються лише одна частина аргументу

f , а інша береться із вхідного даного. Ці міркування і ведуть до наступних визначень

суперпозицій.

Повною (глобальною) суперпозицією функцій g_1, g_2, \dots, g_n в номінативну функцію f

$1 \quad 2 \quad n$

називається функція, яка задається формулою:

$$S(v_1, v_2, \dots, v_n)(f, g_1, g_2, \dots, g_n)(st) = f([v_1 g_1(st), v_2 g_2(st), \dots, v_n g_n(st)]).$$

Частковою (локальною) суперпозицією функцій g_1, g_2, \dots, g_n в номінативну функцію f

$1 \quad 2 \quad n$

називається функція, яка задається формулою:

$$S[v_1, v_2, \dots, v_n](f, g_1, g_2, \dots, g_n)(st) = f(st \nabla [v_1 g_1(st), v_2 g_2(st), \dots, v_n g_n(st)]).$$

Зауважимо, що традиційна суперпозиція в n -арну функцію може розглядатися як частковий випадок повної суперпозиції при тлумаченні $1, \dots, n$ як імен аргументів функції f . Тому

$$S^n(f, g_1, g_2, \dots, g_n) = S^{(1, 2, \dots, n)}(f, g_1, g_2, \dots, g_n).$$

Повернемося тепер до побудови семантичного терму виразу

$$\text{if } N > 1 \text{ then } N * f(N-1) \text{ else } 1$$

Хоча можна обрати повну (глобальну) суперпозицію, візьмемо неповну (локальну) суперпозицію як більш адекватну задачам програмування. Отримуємо семантичний терм

$$\Phi(f) = \text{IF}(S(g_1, N \Rightarrow, 1), S(\text{mult}, N \Rightarrow, S^{[N]}(f, S(\text{sub}, N \Rightarrow, 1))), 1).$$

Неважко довести, що введені композиції суперпозиції є неперервними, тому

теорема КТК є застосовною. Це означає, що існує ННТ $\Phi(f)$, яку можна отримати методом послідовних наближень. Обчислимо декілька таких наближень.

Початковим наближенням є всюди невизначена функція $f_0 = 1$.

Наступне наближення отримуємо підстановкою попереднього наближення в оператор Φ .

$$f^2 = \Phi(f^1) = \Phi(1) = \text{IF}(S(\text{gr}, N \Rightarrow, 1), S(\text{mult}, N \Rightarrow, S(1, S(\text{sub}, N \Rightarrow, 1))), 1) = \text{IF}(S(\text{gr}, N \Rightarrow, 1), S(\text{mult}, N \Rightarrow, 1), 1) = \text{IF}(S(\text{gr}, N \Rightarrow, 1), 1, 1).$$

Наведені рівності мають місце тому, що суперпозиція всюди невизначеної функції, або суперпозиція у всюди невизначену функцію, має своїм результатом

всюди невизначену функцію, тобто $S(1, S(\text{sub}, N \Rightarrow, 1)) = 1$ та $S(\text{mult}, N \Rightarrow, 1) = 1$. Тому суперпозицію часто називають строгою операцією, бо вона зберігає всюди невизначену функцію.

Наближення f^1 можна записати за допомогою її графіка наступним чином

$$f^1([Nan]) = \text{IF}(S(\text{gr}, N \Rightarrow, 1), 1, 1) \quad \square \quad 1, \text{ якщо } n \leq 1,$$

$$f^1([Nan]) = \square \quad \text{невизначено, якщо } n > 1.$$

Обчислимо друге наближення:

$$f^2 = \Phi(f^1) = \text{IF}(S(\text{gr}, N \Rightarrow, 1), S(\text{mult}, N \Rightarrow, S(f^1, S(\text{sub}, N \Rightarrow, 1))), 1) = \text{IF}(S(\text{gr}, N \Rightarrow, 1), S(\text{mult}, N \Rightarrow, S(\text{IF}(S(\text{gr}, N \Rightarrow, 1), 1, 1), S(\text{sub}, N \Rightarrow, 1))), 1).$$

Аналіз отриманого виразу говорить про те, що з'явилося нове значення при $n = 2$. Це значення можна обчислити на $f^2([Na2])$ так, як це ми робили раніше.

Це дає можливість записати наближення f^2 за допомогою його графіка наступним чином

$$f^2([Nan]) = \square \quad 1, \text{ якщо } n \leq 1,$$

$$f^2([Nan]) = \square \quad 2, \text{ якщо } n = 2,$$

$$f^2([Nan]) = \square$$

$$\square \quad i \hat{a} \hat{a} \hat{e} \hat{c} i \hat{a} \hat{a} \hat{i} \quad \hat{i} , \hat{y} \hat{e} \hat{u} \hat{i} \quad n > 2.$$

Наведені наближення дозволяють сформулювати гіпотезу, що

$$\begin{aligned} & \square \quad 1, \hat{y} \hat{e} \hat{u} \hat{i} \quad n \leq 1, \\ & \square \\ f \quad ([N_n]) = & \square \quad n!, \hat{y} \hat{e} \hat{u} \hat{i} \quad 2 \leq n \leq k, \\ & \square \\ & \hat{i} \hat{a} \hat{a} \hat{e} \hat{c} i \hat{a} \hat{a} \hat{i} \quad \hat{i} , \hat{y} \hat{e} \hat{u} \hat{i} \quad n > k. \\ & \square \end{aligned}$$

Цю гіпотезу можна довести індукцією за k.

Для k=1 гіпотеза справедлива.

Доведемо її справедливість для k+1, виходячи з того, що вона справедлива для k.

Маємо, що

$$f_{k+1} = \varphi(f_k) = IF(S(gr, N \Rightarrow, 1), S(mult, N \Rightarrow, S([N]^2, S(sub, N \Rightarrow, 1))), 1)$$

У силу монотонності φ «старі» значення функції f_k не зміняться, але можуть з'явитися нові значення. Це можливо лише для даного $[N_{k+1}]$. Дійсно, обчислення f_{k+1} дає

$$\begin{aligned} f_{k+1}([N_{k+1}]) &= \varphi(f_k) = \\ &= IF(S(gr, N \Rightarrow, 1), S(mult, N \Rightarrow, S([N]^2, S(sub, N \Rightarrow, 1))), 1)([N_{k+1}]) = \\ &= S(mult, N \Rightarrow, S([N]^2, S(sub, N \Rightarrow, 1)))([N_{k+1}]) = \\ &= mult(N \Rightarrow([N_{k+1}]), S([N]^2, S(sub, N \Rightarrow, 1))([N_{k+1}])) = \\ &= mult(k+1, f_k([N_{k+1}])) = \\ &= mult(k+1, f_k([N_{k+1}] \nabla [N_{k+1}])) = \\ &= mult(k+1, f_k([N_{k+1}] \nabla [N_{k+1}])) = \\ &= mult(k+1, f_k([N_{k+1}] \nabla [N_{k+1}])) = \\ &= mult(k+1, f_k([N_{k+1}])) = mult(k+1, k!) = (k+1)!. \end{aligned}$$

Для даних, більших за k+1, значення f_{k+1} буде невизначеним.

Границя послідовності f_1, f_2, f_3, \dots дає функцію f , яка є найменшою

C i

i ∈ ω

нерухомою точкою оператора φ, тобто f = f = lfp φ.
Функція f на даному [Nan] ω C i f

ω

i ∈ ω

приймає значення n! (для n > 0). Таким чином, теорія найменшої нерухомої точки дає обґрунтування рекурсивним визначенням наведеного типу.

Приклад 5.2. Знайдемо ННТ рекурсивного визначення найбільшого спільного дільника чисел n та m. Традиційне рекурсивне визначення має такий вигляд:
f (n, m) = if n > m then f (n-m, m) else if n < m then f (n, m-n) else m

Таке визначення зазвичай трактують як визначення певної бінарної функції. Хоча можна побудувати теорію ННТ і для n-арних функцій (зробіть це самостійно), ми перейдемо до класу номінативних функцій (над даними з іменами M та N). Для цього наше визначення перепишемо у наступному вигляді:
f (N, M) = if N > M then f (N-M, M) else if N < M then f (N, M-N) else M

Побудуємо семантичний терм правої частини цього визначення.

$$\Phi(f) = IF(S (gr, N \Rightarrow, M \Rightarrow), S (f, S (sub, N \Rightarrow, M \Rightarrow), M \Rightarrow), IF(S (less, N \Rightarrow, M \Rightarrow), S (f,$$

$$N \Rightarrow, S (sub, M \Rightarrow, N \Rightarrow)), M \Rightarrow)).$$

Тут gr позначає предикат «більше». Зауважимо, що можна взяти суперпозицію за одним іменем, а саме

$$\Phi'(f) = IF(S (gr, N \Rightarrow, M \Rightarrow), S (f, S (sub, N \Rightarrow, M \Rightarrow)), IF(S (less, N \Rightarrow, M \Rightarrow), S (f, S (sub, M \Rightarrow, N \Rightarrow)), M \Rightarrow)).$$

Також можна брати повну суперпозицію:

$$\Phi''(f) = IF(S (gr, N \Rightarrow, M \Rightarrow), S (f, S (sub, N \Rightarrow, M \Rightarrow), M \Rightarrow), IF(S (less, N \Rightarrow, M \Rightarrow), S (f,$$

$$N \Rightarrow, S (sub, M \Rightarrow, N \Rightarrow)), M \Rightarrow)).$$

Для спрощення перетворень візьмемо варіант для φ'. Початковим наближенням є всюди невизначена функція f = ⊥.

0

Наступне наближення отримуємо підстановкою попереднього наближення в оператор Φ :

$$f_2 = \Phi'(f_1) = \Phi'(\perp) = \text{IF}(S(\text{gr}, N \Rightarrow, M \Rightarrow), S(\perp, S(\text{sub}, N \Rightarrow, M \Rightarrow)), \text{IF}(S(\text{less}, N \Rightarrow, M \Rightarrow), S[M]_1, S[M]_2, \text{IF}(S(\text{gr}, N \Rightarrow, M \Rightarrow), \perp, \text{IF}(S(\text{less}, N \Rightarrow, M \Rightarrow), \perp, M \Rightarrow))), M \Rightarrow)) = \text{IF}(S(\text{gr}, N \Rightarrow, M \Rightarrow), \perp, \text{IF}(S(\text{less}, N \Rightarrow, M \Rightarrow), \perp, M \Rightarrow))$$

Графік задається наступною формулою:

$$f_1([Nan, Mam]) = \begin{cases} \perp & \text{якщо } n = m, \\ \perp & \text{невизначено, якщо } n \neq m. \end{cases}$$

Запишемо графік f_1 трішки іншим чином, фактично вводячи r як найбільший спільний дільник:

$$f_1([Nan, Mam]) = \begin{cases} \perp r, & \text{якщо } (1 \leq r \leq n \wedge 1 \leq r \leq m), \\ \perp & \text{невизначено в інших випадках.} \end{cases}$$

Обчислимо друге наближення.

$$f_2 = \Phi'(f_1) = \text{IF}(S(\text{gr}, N \Rightarrow, M \Rightarrow), S([N]_2, S(\text{sub}, N \Rightarrow, M \Rightarrow)), \text{IF}(S(\text{less}, N \Rightarrow, M \Rightarrow), S([M]_2, S([N]_2, S(\text{sub}, N \Rightarrow, M \Rightarrow))), M \Rightarrow))$$

Його графік задається наступною формулою:

$$f_2([Nan, Mam]) = \begin{cases} \perp & \text{якщо } n = m \text{ або } n = 2m, \\ \perp & \end{cases}$$

$N \Rightarrow), M \Rightarrow)) ([Nan, Mam]) =$

$$\begin{cases} \perp m, & \text{якщо } n = m \text{ або } n = 2m, \\ \perp & \end{cases}$$

= □ n, якщо $2n = m$,
 □
 невизначено в інших випадках.
 □

Пропонуємо читачу побудувати самостійно наступні наближення та довести, що їх границя задає функцію знаходження найбільшого спільного дільника.

Побудуємо тепер розширення мови SIPL, які дозволяють використовувати рекурсивні визначення такого типу, як у розглянутих прикладах.

Синтаксис розширення – його БНФ – подамо у наступній таблиці.

Таблиця 5.1

Ліва частина правила
 Ім'я

Права частина правила

правила
 метазмінна

	<програма > ::=	... program <список об'явлень функцій >
NP1		
NP2		begin <оператор > end
	<вираз > ::=	...
NA1-NA7		
NA8		if <умова > then <вираз > else <вираз >
NA9		<виклик функції >
	<список об'явлень	ε
LDF1	функцій > ::=	<об'явлення функції >
LDF2		<об'явлення функції > ; <список об'явлень
LDF3		функцій >
	<об'явлення	func <ім'я функції > = <вираз >
DF1	функції > ::=	func <ім'я функції > (<список формальних
DF2		параметрів >) = <вираз >
	<ім'я функції > ::=	<змінна >
NF		
	<список формальних	<змінна >
LFP1	параметрів > ::=	<змінна > , <список формальних параметрів >
LFP2		
	<виклик функції > ::=	<ім'я функції >
CF1		
		<ім'я функції > (<список фактичних
CF2		параметрів >)
	<список фактичних	<вираз >
LAP1	параметрів > ::=	<вираз > , <список фактичних параметрів >
LAP2		

У більш математичному вигляді ці розширення можна задати наступним чином:

Таблиця 5.2

Ліва частина правила - Ім'я метазмінна правила	Права частина правила
--	-----------------------

P1	P ::= ...
P2	program dec ₁ ; ... ; dec _n begin S end
A1- A7	...
A8	a ::= if b then a ₁ else a ₂
A9	fc
DF1	dec ::= func fn = a
DF2	func fn (x ₁ , ... , x _n) = a
NF	fn ::= x
CF1	fc ::= fn
CF2	fn (a ₁ , ... , a _n)

Семантичні правила повинні врахувати наявність рекурсивних визначень. Тому ці правила ускладнюються, бо з'являється аргумент E, який називаємо середовищем. Неформально середовище можна трактувати як відображення, яке імені функції fn, заданого рекурсивним визначенням func fn (x₁ , ... , x_n) = a, співставляє формальні

1 n

параметри (x₁ , ... , x_n) та семантику виразу a. Далі ця інформація використовується для

1 n

подання семантики викликів функцій. Деталі тут розписувати не будемо.

Висновки

У цьому розділі було розглянуто основи теорії рекурсивних визначень. Основним

методом знаходження об'єкту, що задається рекурсивним визначенням, є метод послідовних наближень. Щоб цей метод був коректним, клас об'єктів має бути ω -областю (повною частково впорядкованою множиною з найменшим елементом), а операції над об'єктами – неперервними. В цьому випадку теорема Кнастера-Тарського-Кліні гарантує існування найменшого розв'язку (найменшої нерухомої точки неперервного оператора), який знаходиться методом послідовних наближень. Множина нерухомих точок також утворює ω -область, а відображення, яке неперервному оператору співставляє його найменшу нерухому точку, також є неперервним. Традиційні методи побудови більш складних областей дозволяють будувати нові ω -області з більш простих ω -областей. Застосування теорії найменшої нерухомої точки є надзвичайно широким. Зокрема, ця теорія може бути застосована для визначення синтаксису формальних мов за допомогою БНФ та подібних методів. Семантика рекурсивних програм також може бути визначена за допомогою цієї теорії. Важливо відзначити, що семантичні області є ω -областями, а засоби конструювання програм – композиції – є неперервними. Ці властивості гарантують наявність адекватного тлумачення рекурсивних програм.

Контрольні питання

1. Що таке рекурсивне визначення (рекурсивне рівняння)?
2. Які парадокси пов'язані з рекурсивними визначеннями?
3. Який метод застосовують для розв'язку рекурсивних рівнянь? Які поняття пов'язані з цим методом?
4. Що таке перший граничний (перший нескінченний) ординал?
5. Що таке ω -область?
6. Дайте визначення неперервного відображення.
7. Сформулюйте теорему Кнастера-Тарського-Кліні.
8. Яку структуру має множина нерухомих точок неперервного оператора?
9. Визначте методи конструювання похідних областей.
10. Сформулюйте властивості оператора найменшої нерухомої точки.
11. Сказайте на застосування теорії ННТ для подання синтаксису мов програмування.
12. Вкажіть на застосування теорії ННТ для подання семантики мов програмування.
13. Як визначаються рекурсивні розширення мови SIPL.
14. Наведіть приклади рекурсивних визначень у розширеннях мови SIPL.

Вправи

1. Розбудуйте теорію ННТ для класу n -арних функцій.
2. Для задач з розділу 1 побудуйте рекурсивні визначення.
3. Доведіть коректність побудованих рекурсивних визначень.

Література:

1. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции (Том 1. Синтаксический анализ). – М.: Мир, 1978. – 612 с.

2. Басараб И.А., Никитченко Н.С., Редько В.Н. Композиционные базы данных. - К.: Либідь, 1992. - 182 с.
3. Глушков В.М., Цейтлин Г.Е., Ющенко Е.Л. Алгебра. Языки. Программирование.- К.: Наукова думка, 1974. - 328 с.
4. Грис Д. Наука программирования. - М.: Мир, 1982.
5. Гросс М., Лантен А. Теория формальных грамматик.- М.: Мир, 1971.- 294 с.
6. Дейкстра Э. Дисциплина программирования. - М.: Мир, 1976.
7. Лавров С. Программирование. Математические основы, средства, теория. - СПб.: БХВ-Петербург, 2001. - 320 с.
8. Нікітченко М.С., Шкільняк С.С. Математична логіка та теорія алгоритмів. - К., 2008. - 528 с.
9. Пентус А.Е., Пентус М.Р. Теория формальных языков: Учебное пособие. - М.: Изд-во ЦПИ при механико-математическом ф-те МГУ, 2004. - 80 с.
10. Редько В.Н. Композиции программ и композиционное программирование // Программирование. - 1978. - N.5. - С. 3-24.
11. Хопкрофт Дж., Мотвани Р., Ульман Д. Введение в теорию автоматов, языков и вычислений. - М.: Вильямс, 2002.- 528 с.
12. Abrial J.-R. Assigning Programs to Meanings.- Cambridge University press. - 1996. - 779 p.
13. Hoare C.A.R., Jifeng H. Unifying Theories of Programming. - Prentice Hall Europe. - 1998. - 298 p.
14. Nielson H.R., Nielson F. Semantics with Applications. A Formal Introduction.- John Wiley & Sons, Chichester, England, 1992. - 240 p.
15. Nikitchenko N. A Composition Nominative Approach to Program Semantics. - Technical Report IT-TR: 1998-020. - Technical University of Denmark. - 1998. - 103 p.
16. Spivey J.M. Understanding Z: A specification language and its formal semantics. - Cambridge University press. - 1988. - 131 p.
17. Winskel G. The Formal Semantics of Programming Languages: An Introduction. - The MIT Press, London: England, 1993. - 361 p.

ЗМІСТ

ВСТУП.....	3
1. Формалізація простої мови програмування.....	6
1.1. Неформальний опис простої мови програмування.....	6
1.2. Формальний опис синтаксису мови SIPL.....	7
1.3. Формальний опис семантики мови SIPL.....	12
1.3.1. Дані.....	12
1.3.2. Функції.....	13

1.3.3.	
Композиції.....	1
1.3.4. Програмні алгебри.....	15
1.3.5. Визначення семантичних термів.....	17
1.3.6. Побудова семантичного терму програми.....	18
1.3.7. Обчислення значень семантичних термів.....	20
1.3.8. Загальна схема формалізації мови SIPL.....	21
1.4. Властивості програмної алгебри.....	22
2. Розвиток основних понять програмування.....	31
2.1. Аналіз словникових визначень поняття програми.....	31
2.2. Розвиток поняття програми з гносеологічної точки зору.....	32
2.3. Розвиток основних понять програмування.....	36
2.3.1 Початкова тріада понять програмування.....	37
2.3.2. Тріада прагматичності програм.....	39
2.3.3. Тріада основних понять програмування.....	40
2.3.4. Пентада основних понять програмування.....	41
2.4. Розвиток основних програмних понять.....	43
2.4.1. Тріада основних програмних понять.....	43
2.4.2. Пентада основних програмних понять.....	46
2.5. Сутнісні та семіотичні аспекти програм.....	48
2.6. Програми і мови.....	51
2.7. Пентада програмних понять процесного типу.....	51
3. Формалізація програмних понять.....	5
3.1. Теоретико-функціональна формалізація.....	5
3.2. Класи функцій.....	55
3.3. Програмні системи.....	55
3.4. Рівні конкретизації програмних систем.....	57
4. Синтактика: формальні мови та граматики.....	60
4.1. Розвиток понять формальної мови та породжуючої граматики.....	60
4.2. Визначення основних понять формальних мов.....	66
4.3. Операції над формальними мовами.....	67

4.4. Породжуючі граматики	68
4.5. Приклад породжуючої граматики та її властивості	70
4.6. Ієрархія граматик Хомського	72
4.7. Автоматні формалізми сприйняття мов	75
4.7.1. Машини Тьюрінга	75
4.7.2. Еквівалентність машин Тьюрінга та породжуючих граматик	78
4.7.3. Лінійно-обмежені автомати	80
4.7.4. Магазинні автомати	80
4.7.5. Скінченні автомати	81
4.8. Методи подання синтаксису мов програмування	82
4.8.1. Нормальні форми Бекуса-Наура	83
4.8.2. Модифіковані нормальні форми Бекуса-Наура	83
4.8.3. Синтаксичні діаграми	8
4.9. Властивості контекстно-вільних граматик	86
	120
----- Page 121-----	
4.9.1. Видалення несуттєвих символів	86
4.9.2. Видалення ϵ -правил	88
4.9.3. Нормальна форма Хомського	89
4.9.4. Нормальна форма Грейбах	89
4.9.5. Рекурсивні нетермінали	89
4.10. Властивості контекстно-вільних мов	90
4.11. Операції над формальними мовами	90
4.12. Дерева виводу	92
4.13. Однозначні та неоднозначні граматики	93
4.14. Розв'язні та нерозв'язні проблеми КВ-грамматик та мов	9

4.15. Рівняння в алгебрах формальних мов.....	95
5. Теорія рекурсії (теорія найменшої нерухомої точки)	100
5.1. Рекурсивні визначення та рекурсивні рівняння.....	100
5.2. Частково впорядковані множини, границі ланцюгів та ω -області.....	101
5.3. Неперервні відображення	102
5.4. Теореми про нерухомі точки	102
5.5. Конструювання похідних ω -областей.....	10
5.6 Властивості оператора найменшої нерухомої точки.....	107
5.7. Застосування теорії ННТ	108
5.7.1. Уточнення синтаксису мов програмування.....	108
5.7.2. Семантика мов програмування.....	110
5.7.3. Рекурсивні розширення мови SIPL.....	113