Семантичні моделі вивчення програмування

Постановка задачі

Програмування P_{asl} розглядаємо як трикроковий процес: аналіз задачі (analysis), будова алгоритму розв'язування (solve), реалізація алгоритму визначеною алгоритмічною мовою (language): $P_{asl} = \{A_p, S_a, L_r\}$. Аналіз задачі A_p залежить від предметної області, яку потрібно співставити з операціями і функціями середовища програмування комп'ютера, і побудувати початкове відображення $A_p \rightarrow C_{op}$, де C_{op} – базові операції і функції комп'ютера. Будову алгоритму S_a виконуємо в термінах або послідовного покрокового виконання операцій, або розподілених обчислень на основі асинхронного опрацювання подій. Реалізація L_r алгоритмічною мовою враховує синтаксичні і семантичні особливості конкретної мови, а також вибір потрібних структур даних.

Моделі другого кроку S_a будуємо на основі варіанта денотативної семантики, описаної далі. Для розробки алгоритмів треба точно знати порядок виконання і особливості кожного елемента алгоритмічної мови. На основі таких знань розробка алгориму стає продуктивнішою.

Будемо розглядати будову моделей вивчення програмування на прикладі мови C++.

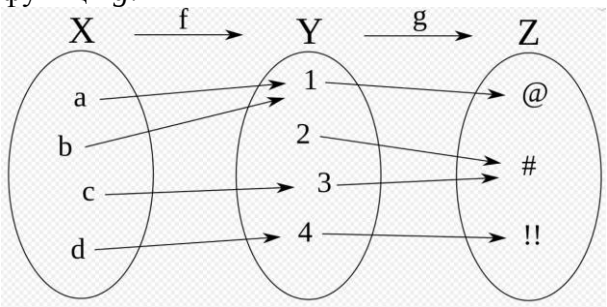
Денотативна семантика

[https://uk.wikipedia.org/wiki/Семантика_мов_програмування]

Денотати́вна сема́нтика — це підхід до формалізації семантики програмних систем за допомогою математичних об'єктів, які описують зміст системи. Кожній конструкції мови програмування відповідає якась математична інтерпретація. Опис системи у денотативній семантиці здійснюється за допомогою математичних об'єктів та абстрагований від її реалізації конкретною мовою програмування.

Важливим аспектом денотативної семантики є принцип композиційності, за яким денотат програми будується з денотатів її складових частин за допомогою операції композиції. Компози́ція (суперпозиція) фу́нкцій (відображень) в математиці – функція, побудована з двох функцій таким чином, що результат першої функції є аргументом другої.

Наприклад, композиція функцій f: X→Y та g: Y→Z будується так: аргумент x з X застосовується до першої функції f, а її результат y з Y застосовується як аргумент до другої функції g:



Методи денотативної семантики базуються на відповідних алгебрах.

Надалі будемо розглядати варіант денотативної семантики, за яким будуємо моделі семантики на основі класифікації операцій і функцій перетворення даних. Моделі трактуємо як універсальні алгебри U(A)=<M;Ω>, де M – деяка непуста множина (величин, комірок пам'яті, структур, операторів), а Ω - сукупність операцій (можливо частинних) на множині M, включаючи сигнатуру.

Синтаксис підмножини мови

Будемо використовувати два способи зображення синтаксису.

1. Нотація Бекуса-Наура. Класичний спосіб зображення синтаксису.

Продукцією або *правилом підстановки* називається впорядкована пара (U,x), яка записується

`U ::= x ,`

де U – символ, x – непустий скінченний ланцюжок символів. U – ліва частина, x – права частина правила підстановки.

Граматикою G[Z] називається скінченна непуста множина правил; Z – символ, який повинен зустрітися у лівій частині по крайній мірі одного правила. Він називається початковим символом або *аксіомою*. Всі символи, які зустрічаються у лівих і правих частинах правил утворюють *словник* V.

Нотація БНФ є набором «продукцій», кожна з яких відповідає зразку `U::=x` .

`<` *лівий обмежувач виразу*

`>` *правий обмежувач виразу*

`::=` *визначене як*

`|` *або*

Ці чотири символи є символами мета-мови, вони не належать мові, котру описують.

2. Розширена форма Бекуса-Наура.

Є багато різновидів нотації Бекуса-Наура специфікації граматик алгоритмічних мов. Зокрема, застосовують правила будови регулярних виразів, представлень окремих символів, квантифікатори, групування, перерахування. Часто використовують такі позначення:

`::=` *визначене як*;

`|` *або*;

`idword` *комбінація букв, цифр і знаків є позначенням нетермінального символа*;

`"symbols"` *символи в лапках є термінальними, які безпосередньо записують в тексті*;

`( )` *круглі дужки використовують для визначення області дії; шаблон усередині групи обробляється як єдине ціле й може бути квантифікованим*; *квантифікатор після символу або групи визначає, скільки разів попередній вираз може зустрічатися*;

`*` *кількість повторень нуль або більше*;

`+` *кількість повторень одне або більше*;

`?` *кількість повторень нуль або одне*;

`[ ]` *область дії, кількість повторень нуль або одне*;

Приклад. Граматика довільного числа і підмножини виразів з деякими спрощеннями.

а) Класична нотація Бекуса-Наура.

```
<чс> ::= < число >  |  + < число >  |  - < число >
<число> ::= <ціле> | <фіксована крапка> | <експоненціальний формат>
<ціле>  ::=  цифра  |  <ціле> цифра
<фіксована крапка>  ::=  <ціле> . <ціле>
<експоненціальний формат>  ::=  <мантиса> <порядок>
<мантиса> ::=  <ціле>  |  <фіксована крапка>
<порядок> ::=  E <знак> <ціле>  |  E <ціле>
<знак> ::=  +  |  -
<вираз> ::= <доданок> | <вираз> + <доданок> | <вираз> - <доданок>
<доданок> ::= <множник> | <доданок> * <множник> | <доданок> /
              <множник> | <доданок> % <множник>
<множник> ::= змінна | <число> | <функція>() | (<вираз>)
<функція> ::= abs | rand | srand
```

б) Розширена форма Бекуса-Наура.

```
чс ::= [ "+" | "-" ] число
число ::= ціле | фіксована_крапка | експоненціальний_формат
ціле  ::=   "цифра" +
фіксована_крапка  ::=   ціле "." ціле
експоненціальний_формат  ::=   мантиса  порядок
мантиса ::=  ціле  |  фіксована_крапка
порядок ::=  "E" [ знак ] ціле
знак ::=   "+" | "-"
вираз ::= доданок ( знак доданок )*
доданок ::= множник ( ( "*" | "/" | "%" ) множник )*
множник ::= "змінна" | число | функція "()" | "(" вираз ")"
функція ::= "abs" | "rand" | "srand"
```

3. Підмножина C++ (розширена форма Бекуса-Наура).

```
знак ::=   "+" | "-"
вираз ::= доданок ( знак доданок )*
доданок ::= множник ( ( "*" | "/" | "%" ) множник )*
множник ::= "змінна" | число | функція "()" | "(" вираз ")"
функція ::= "abs" | "rand" | "srand"
оператор ::= присвоєння | читання | друкування |
           умовний | цикл_з_параметром | цикл_з_передумовою
читання ::= "cin" ">>" "змінна" (">>" "змінна")*  ";"
друкування ::= "cout" "<<" вираз ("<<" вираз)*  ";"
присвоєння ::= "змінна" "=" вираз ";"
умовний ::= "if" "(" умова ")" блок "else" блок |
            "if" "(" умова ")" блок
блок ::= оператор ";"  | "{" ( оператор ";" )* "}"
цикл_з_параметром ::= "for" "(" правило_параметра ")" блок
цикл_з_передумовою ::= "while" "(" умова ")" блок
```

Зауважимо, що для наших цілей не обов'язково формально визначати підмножину алгоритмічної мови щодо всіх елементів. Так, поняття `умова` або `правило_параметра` можна не уточнювати на цьому етапі навчання. Крім того, поняття `оператор` не зобов'язане бути повним щодо цілої алгоритмічної мови.

Денотативна семантика для навчання C++ на рівні моделей

Моделі семантики визначаємо двома групами: 1) моделі на основі допустимих операцій мови C++; 2) моделі на основі операторів і керуючих структур мови C++. Для першої групи моделей будуємо алгебри виду $U_A^i=<t_i; \{op_{ti}\}>$, де t_i – тип програмного об'єкта C++, op_{ti} – визначені операції для t_i. Для другої групи моделей будуємо алгебри виду $U_A^i=<\{par\}_{si}; \{operator_{si}: read,write\}>$, де $\{par\}_{si}$ – параметри виконання оператора або структури si, $\{operator_{si}: read,write\}$ – функція перетворення вхідних значень або станів об'єктів (величин) read у вихідні значення або стани write.

Далі подаємо для демонстрації варіанти моделей семантики окремих елементів мови C++. Підкреслимо, що подаємо лише самі моделі семантики, але не повне визначення семантики. Моделі семантики потрібні, щоб обрати елементи мови і фокусувати на них увагу. Для обраних елементів мови будуємо приклади для учнів.

Моделі програмування на основі операцій. Цілочисельна арифметика

Багатокрокова послідовна схема вивчення програмування ділиться на дві частини. Перша частина враховує особливості внутрішніх типів даних виду integer→ real→ [byte,longint]→ boolean→ char→ enum→ string→ record→ set of→ file of→ pointer. Для кожного типу t_i будуємо алгебру U_A^i=<t_i;{op_{ti}}>, де op_{ti} – визначені операції для типу t_i.

Вивчення розділу цілочисельних типів даних будуємо в такому порядку.

За відомими означеннями стандарта мови С++ записуємо підмножину алгоритмічної мови щодо використання цілих типів даних в нотації металінгвістичних формул, яка подана вище в пп. "Підмножина С++ (розширена форма Бекуса-Наура)".

На основі металінгвістичних формул визначаємо алгебру U_A^{int}=<int; {+, -, *, /, % ,=, abs, rand, srand}>. Позначимо Ω= {+, -, *, /, %, =, abs, rand, srand} сукупність операцій для типу int. Тоді виконуємо розділення U_A^{int} на частинні випадки шляхом виділення підмножин операцій, враховуючи спорідненість операцій, за такою схемою:

<int; {Ω^1}> → <int; {Ω^2}> → <int; {Ω^3}> → ..., де $\Omega^1 \subset \Omega^2 \subset \Omega^3 \subset ... \subset \Omega^n$.

Зауважимо, що таке розділення якраз є визначальним для подальшої побудови схеми навчання. Побудова частинних випадків шляхом додавання окремих операцій

<int; {+}> → <int; {+, -}> → <int; {+, -, *}> → і т.д.

є непродуктивною, бо зростає загальний обсяг прикладів і задач і виникає дублювання на змістовному рівні задач.

Для випадку алгоритмічної мови С++ будуємо схеми навчання, приймаючи до уваги підмножини типів, операцій і функцій.

1) Базові типи і операції (рис.1).

U_{A1}^{int}= < int, short, long; +, - , *, / >

Отже, обираємо початковий перелік типів даних і початковий перелік допустимих операцій. Сам по собі такий запис є лише моделлю семантики і показує належність елементів мови до семантики, але не дає визначення семантики. Вважаємо, що визначення семантики в формі довільного пояснення будуть подані окремо.

Рис.1. Базові типи і операції

У цьому випадку схему починаємо будувати справа наліво: спочатку записуємо мовою С++ вирази, але не обов'язково повні оператори, відновлюємо математичний запис формули MF (обернена задача), і формулюємо задачу SZ.

2) Ділення по модулю, присвоєння, багатокрокові обчислення (рис.2).

U_{A2}^{int}= < int, short, long; +, -, *, /, %, = >
$U_{A2=}^{int}$= <{a,b,c}= ; { =: read(a,b), write(c) }>

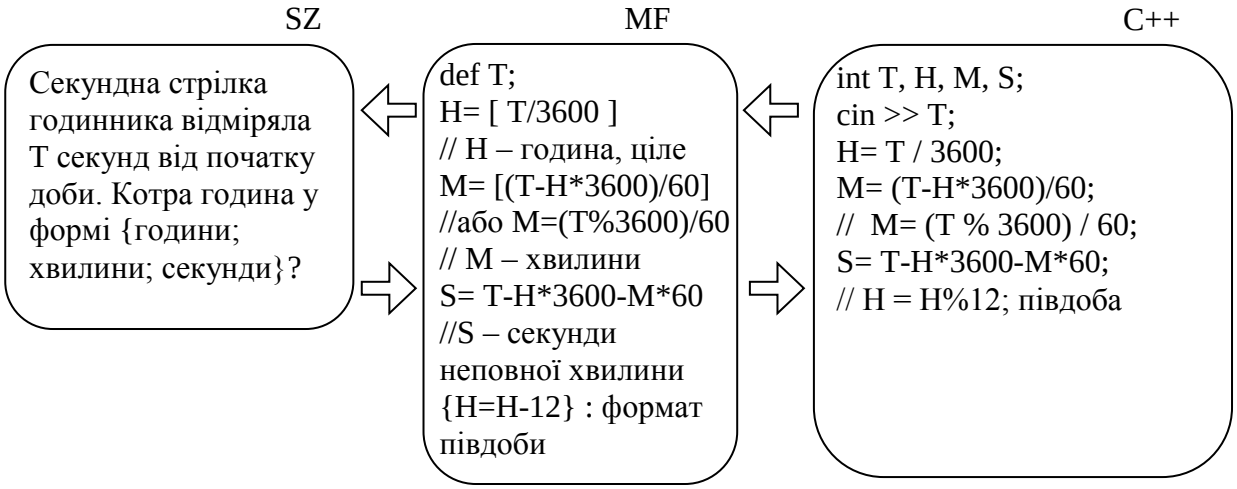


Рис.2. Ділення по модулю, присвоєння, багатокрокові обчислення

У таких випадках схема (SZ, MF, C++) може бути ітераційною.

3) Частинні операції – обчислення при переході за гранично допустиму межу (рис.3).

U_{A3}^{int}= < short; +, -, *, = >

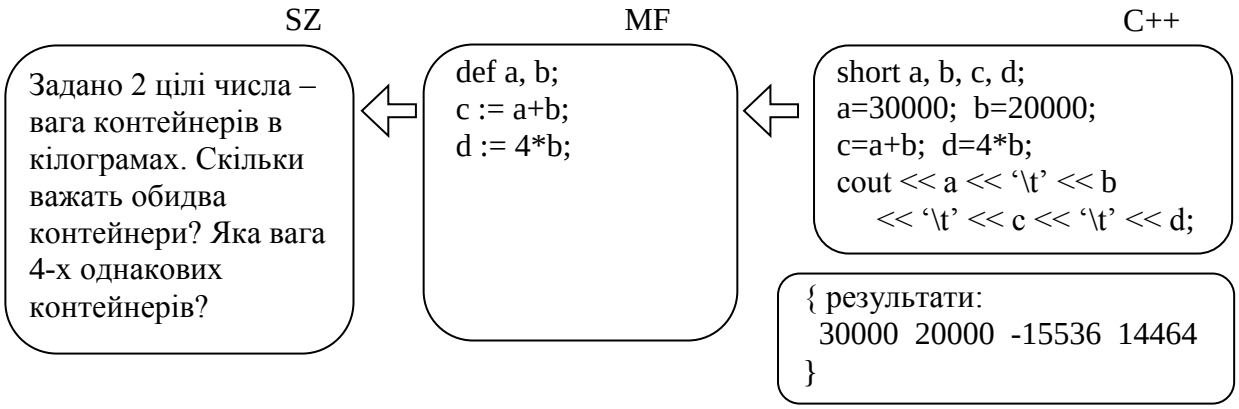


Рис.3. Перехід за гранично допустиму межу

4) Знакові і беззнакові величини (рис.4).

U_{A4}^{int}= < int, short, long, unsigned; +, -, *, /, %, = >

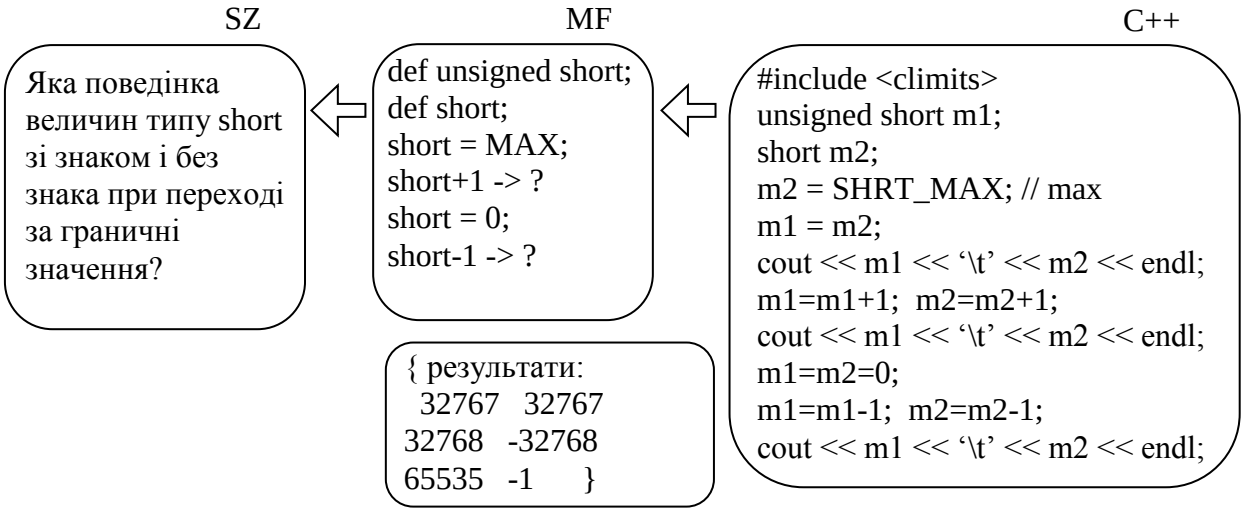


Рис.4. Знакові і беззнакові величини

5) Математичні функції – різні групи функцій (рис.5, *а*, *б*).

U_{A5}^{int}= < int, short, long; +, -, *, /, %, =, abs, rand, srand >

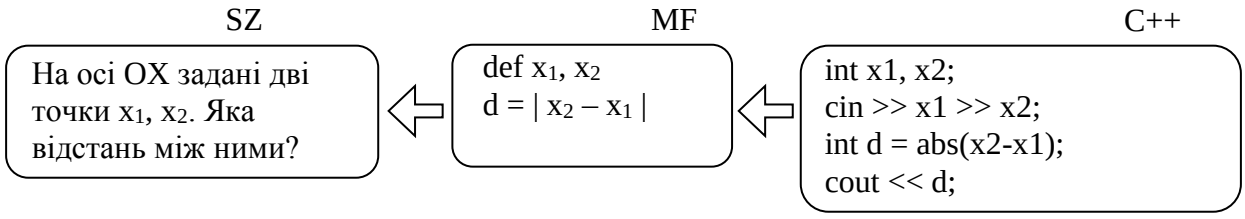


Рис.5,*а*. Математичні функції

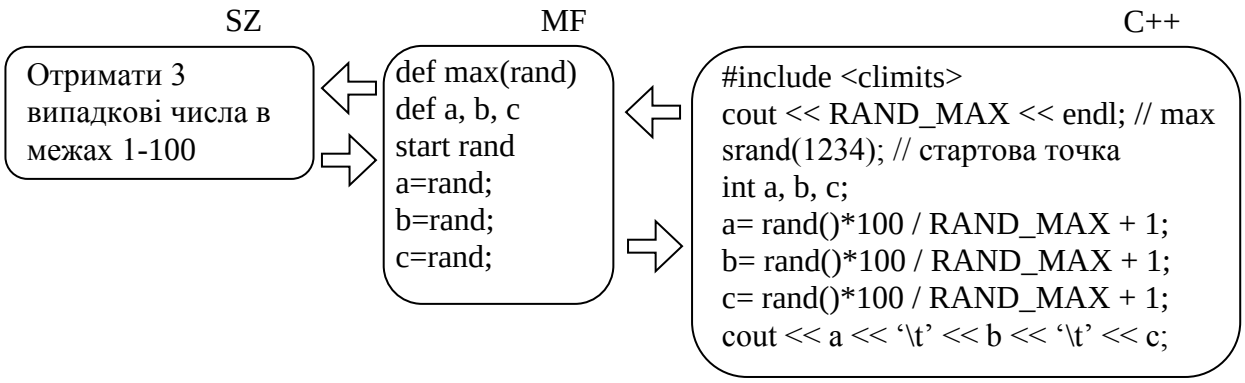


Рис.5,*б*. Математичні функції

Зауваження. Для випадку дійсних типів чисел моделі будуємо на основі функцій, які за своїм змістом обчислюють результат як дійсне число (рис.5,*в*).

$U_{A5\text{-}1}^{double}$= < int, short, long, double; +, -, *, /, %, =, sin, cos, tan >
$U_{A5\text{-}2}^{double}$= < int, short, long, double; +, -, *, /, %, =, sin, cos, tan, sqrt, exp, log, pow >

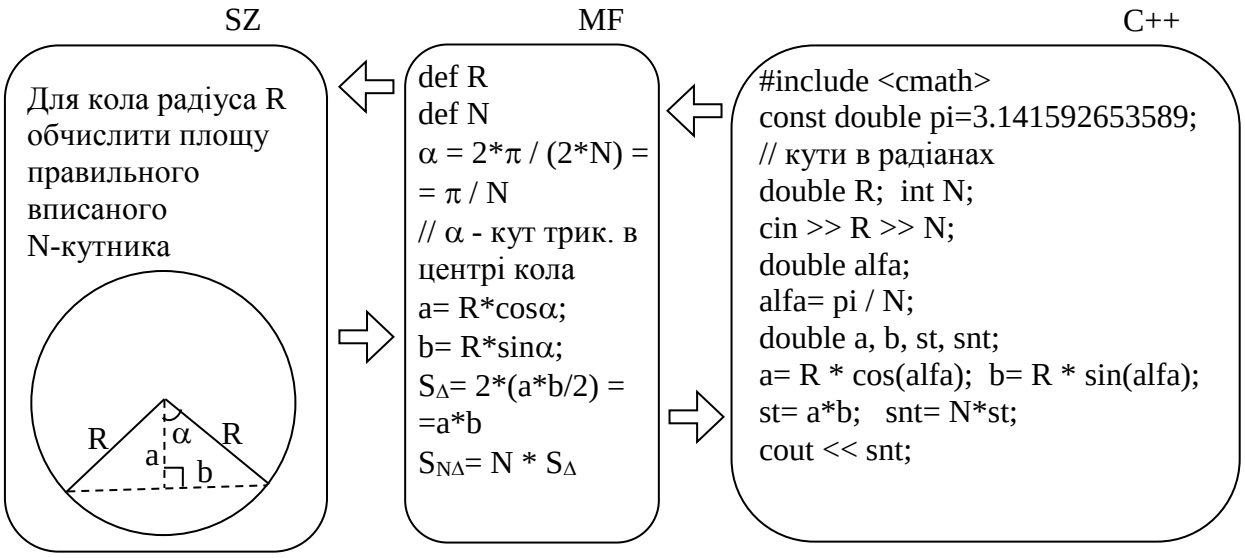


Рис.5,*в*. Функції для дійсних чисел

Моделі програмування на основі операторів і керуючих структур

Друга частина загальної схеми вивчення програмування враховує функціональні перетворення даних на рівні операторів і керуючих структур: (:=) → if then → if then else → for → while → repeat → for(init;cont;modif) → do while → case, switch → break, continue → function, procedure → class → unit. Для кожного типу оператора або структури будуємо алгебру U_A^i=<{memory value}$_{ti}$; {operator$_{ti}$: read, write}>, де {memory value}$_{ti}$ – сукупність комірок пам'яті (величин), причетних до виконання оператора ti, а {operator$_{ti}$: read, write} – функція перетворення значень комірок пам'яті до виконання оператора (read) в результуючі значення (write).

6) Умовний оператор і його застосування (рис.6).

$U_{A6\text{-}if}^{int}$= < {a, b, c, d}$_{if}$; { if: read(a,b), write([c,d]=f_1(a,b), [c,d]=f_2(a,b)) } >

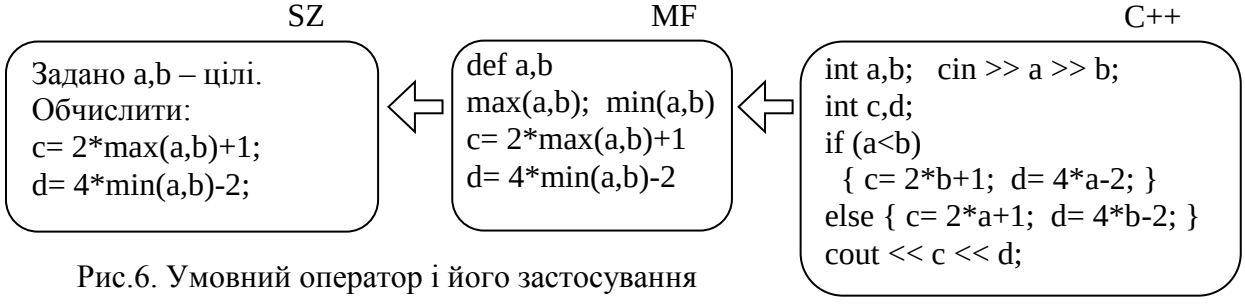


Рис.6. Умовний оператор і його застосування

В загальній формі умовний оператор можна визначити так:

$U_{A6\text{-}if^*}^{int}$= < {a_1,a_2,...,a_k}$_{if}$; { if: read({a_r}⊆{a_1,...,a_k}), write((f_1,f_2)({a_w}⊆{a_1,...,a_k})) } >

де f_1,f_2 – дві різні функції обчислення результуючих значень для умов true і false. До такого визначення умовного оператора будуємо задачі на функціональні перетворення даних за деяким параметром умови.

7) Цикл загальної форми for (рис.7).

$$U_{A7\text{-}for}^{int} = < \{a, b, c, d\} \cup \{x_i\}_{for} ; \{ for: read(a,b,\{x_i\}), write(c,d,\{x_i\}) \} >$$

де a,b,c,d – скалярні значення; x_i – масив (масиви, вектори).

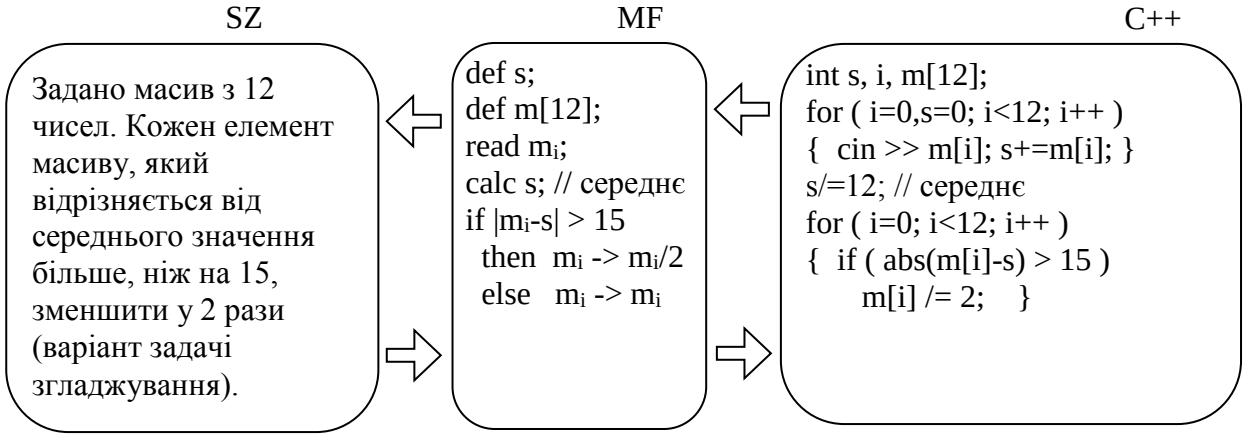


Рис.7. Цикл загальної форми for

8) Цикл загальної форми while.

$$U_{A8\text{-}while}^{int} = < \{a,b,c,d,vbool\} \cup \{x_i\}_{while} ; \{ while: read(a,b,vbool,\{x_i\}), write(c,d,vbool,\{x_i\}) \} >$$

де a,b,c,d – скалярні значення; vbool – логічне значення величини або виразу для обчислення умови продовження циклу; x_i – масив (масиви, вектори). Особливістю циклу while є величина vbool, значення якої потрібно міняти при кожному виконанні тіла циклу. Така величина (або декілька величин) присутня в умові продовження циклу.

а) до такої алгебри повторюємо задачі п.7 для порівняння реалізації різними операторами циклу.

б) будуємо задачі з доцільною реалізацією оператором форми while (рис.8).

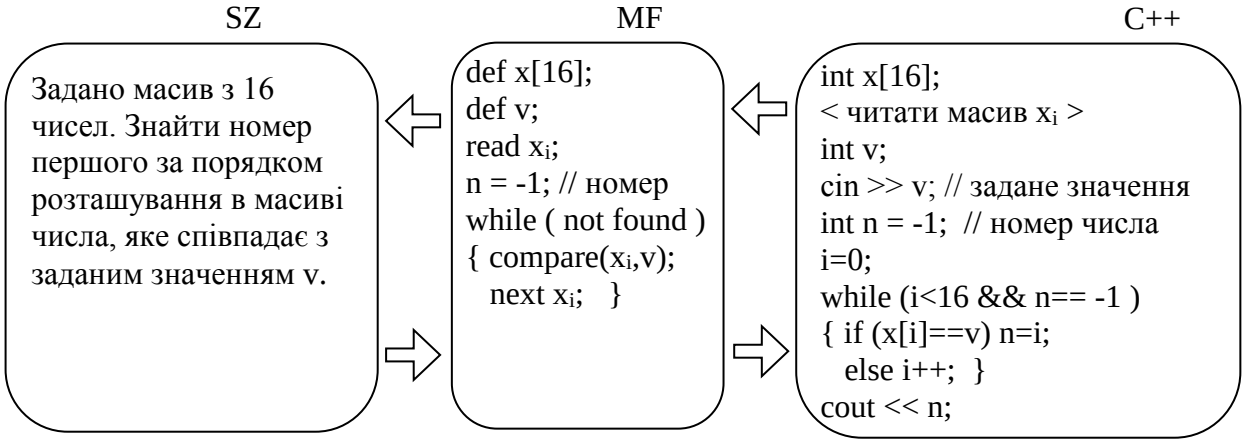


Рис.8. Задачі для оператора while

9) Комбінація циклу типу for (або while) і умовного оператора if.

$$U_{A9\text{-}for+if}^{int} = < \{min, max\} \cup \{x_i\}_{for+if} ; \{ for: read(x_i), write()\}; if: read(min,max,x_i), write(min,max) \} >$$

Така алгебра означає перебір (читання) елементів масиву x_i на основі оператора for, перевірку оператором if елементів масиву x_i і деяких величин min, max, і запис (повторне обчислення) оператором if величин min, max. Очевидний приклад задачі на реалізацію цієї алгебри: у заданому масиві чисел знайти найменше і найбільше значення.

10) Вкладені цикли for+for або while+while, комбінація вкладених циклів і умовного оператора for+for+if.

$$U_{A10\text{-}for+for}^{int} = < \{m_{ij}\} \cup \{s_i\}_{for+for} ; \{ for+for: read(m_{ij}), write(s_i) \} >$$

Використовуємо матрицю чисел m_{ij} і масив s_i. Комбінація вкладених циклів забезпечує перебір (читання) елементів матриці m_{ij} і обчислення елементів масиву s_i. Приклад задачі: обчислити суму елементів кожного рядка матриці. Подібно будуємо алгебру $U_{A10\text{-}for+for+if}^{int}$ для задач виду: обчислити мінімальний елемент кожного рядка заданої матриці.