Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут»

В. М. Медведєва
В. А. Третяк

Транслятори: внутрішнє подання програм та інтерпретація

Рекомендовано Вченою радою НТУУ «КПІ»
як навчальний посібник для студентів, які
навчаються за напрямами підготовки
«Програмна інженерія», «Комп'ютерні науки»

Київ
НТУУ «КПІ»
2015

УДК 004.4’42(075.8)
ББК 32.973-018.2я73
М42

Гриф надано Вченою радою НТУУ «КПІ»
(протокол № 2 від 02.03.2015 р.)

Рецензенти:
Ю. І. Бадаєв, д-р техн. наук, проф.,
Київська державна академія водного транспорту
В. Г. Писаренко, д-р фіз.-мат. наук, проф.,
Інститут кібернетики ім. В. М. Глушкова НАН України

Відповідальний редактор
С. О. Лук’яненко, д-р техн. наук, проф.,
Національний технічний університет України
«Київський політехнічний інститут»

Медведєва, В. М.

М42 Транслятори: внутрішнє подання програм та інтерпретація [Текст] : навч. посіб. / В. М. Медведєва, В. А. Третяк. – К. : Текст, 2015. – 144 с. – Бібліогр. : С. 141-143. – 150 пр.

Викладено другу частину дисциплін «Основи розробки трансляторів» та «Лінгвістичне забезпечення САПР». Розглянуто способи внутрішнього подання програм, організації таблиць, а також деякі нюанси розподілу пам’яті в трансляторах. Наведено приклади та завдання для самостійної роботи студентів.

Для студентів напрямів підготовки «Комп’ютерні науки» та «Програмна інженерія», може бути використаний студентами інших спеціальностей.

УДК 004.4’42(075.8)
ББК 32.973-018.2я73

ЗМІСТ

ВСТУП

Дана книга є продовженням навчального посібника «Транслятори: лексичний та синтаксичний аналізатори» [1]. В попередній частині розглянуто методи виконання лексичного та синтаксичного аналізу, що є початковими етапами трансляції.

У сучасних трансляторах після поділу тексту вихідної програми, що надходить на вхід транслятора, на лексеми лексичним аналізатором, виконується перевірка правильності слідування лексем синтаксичним аналізатором. В разі, якщо вихідна програма написана вірно з точки зору синтаксису, виконується переведення коду вихідної програми в проміжну форму подання. В деяких трансляторах переведення програми в проміжну форму може виконуватися одночасно з синтаксичним розбором. Після цього здійснюється виконання програми інтерпретатором або побудова внутрішнього подання програми компілятором.

В посібнику розглядаються способи проміжного подання програм, алгоритми їх отримання та інтерпретації, особливості організації пам’яті компілятора. Перший розділ присвячено формам подання граматик, описуються способи побудови польського інверсного запису під час висхідного синтаксичного аналізу та окремо за алгоритмом Дейкстри, а також способи переведення програми у послідовність тріад або тетрад. У другому розділі основна увага приділяється організації пам’яті під час інтерпретації, детально розглядаються способи статичного та динамічного розподілу пам’яті.

Знання та вміння щодо трансляції коду необхідні при розв’язанні багатьох прикладних задач, насамперед, в гібридних програмних комплексах, компоненти яких базуються на різних концепціях програмування. Так, набуті вміння використовуються при створенні програмного інструментарію обміну повідомленнями між ГІС-системою та ядром штучного інтелекту в програмному комплексі моніторингу гідрохімічного стану підземних вод АЕС (НДР № державної реєстрації 0115U000329).

Хоча матеріал базується на строгій математичній теорії, він викладається у вільній формі, тому може бути зрозумілим читачам, що не звикли до математичних формулювань. Посібник щедро оздоблений ілюстративним матеріалом: різноманітними схемами та таблицями, – а також прикладами, які детально обговорюються та дозволяють краще зрозуміти практичну цінність описаних алгоритмів. Наведений матеріал має практичне спрямування, для забезпечення можливості програмної реалізації зазначених методів читачами. Однак з метою узагальнення та надання читачам вільного вибору засобів та способів реалізації в посібнику не наводяться приклади реалізації за допомогою існуючих мов програмування.

Такий стиль викладення обумовлений цільовою аудиторією навчального посібника, яку складають студенти, майбутні фахівці в галузі інформатики та обчислювальної техніки. Він містить виклад третього та четвертого розділів дисциплін «Основи розробки трансляторів» та «Лінгвістичне забезпечення САПР». Зазначені дисципліни включені до циклу «Професійної підготовки» варіативної частини навчальних планів підготовки бакалаврів з напряму «Програмна інженерія» спеціальностей «Програмне забезпечення систем» та «Інженерія програмного забезпечення», а також напрямку «Комп'ютерні науки» спеціальності «Інформаційні технології проектування».

Наведена інформація дозволить студентам краще зрозуміти загальні властивості роботи трансляторів в цілому і, як наслідок, тих засобів розробки програмного забезпечення, якими вони користуються чи будуть користуватися в професійній діяльності.

ПРИЙНЯТІ ПОЗНАЧЕННЯ

Через велику кількість прикладів різноманітного коду в тексті посібника не використовуються традиційні лапки для позначення частин коду, натомість елементи вхідного коду та проміжних форм подання коду виділяються жирним шрифтом. Наприклад, вираз «b+a/2» в тексті посібника буде записаний так: **b+a/2**.

Приклади програм на псевдо-коді оформлюються як рисунки наступного вигляду:

```
function Z(var Zsem : Semantica): Boolean;
begin
  тіло функції
end;
```

Псевдо-код має нотацію подібну до Delphi. Сам псевдо-код будується таким чином, щоб лише показати алгоритм, а вибір деталей реалізації покладається на читача. Деякі операції, що залежать від обраних користувачем структур даних, записуються як звичайний текст, такий текст виділяється курсивом. В текстових описах псевдо-коду назви наведених в ньому функцій та змінних виділяються сірим кольором, наприклад, в наведеному вище прикладі описано функцію Z, що приймає параметр Zsem типу Semantica.

Важливі зауваження виділяються рамкою.

В кінці кожного розділу наводяться завдання для самоконтролю. Перед завданнями зображено студента, що натхненно і зосереджено пише транслятор своєї мови програмування:

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

ПОЛІЗ – польський інверсний запис;

УПХ – умовний перехід по хибності;

БП – безумовний перехід;

ЗПЦ – змінна параметра циклу;

ОЦ – ознака циклу;

ЛРК – лічильник робочих міток;

БПВ – безумовний перехід з поверненням;

В – повернення, що забезпечує резервування комірки ПОЛІЗу (для команди повернення, що генерується операцією БПВ);

ОТЕ – ознака типу елемента циклу зі списком елементів;

ВС – вказівник стеку;

РБ – рівень блока;

ВПБ – вказівники на початки блоків.

ПФ – процедура-функція

РОЗДІЛ 1. ПРОМІЖНІ ФОРМИ ПОДАННЯ ПРОГРАМИ

В процесі генерації об'єктного коду початкова вхідна програма переводиться в деяку внутрішню форму подання, зручнішу для подальшої обробки та оптимізації. Форми внутрішнього подання поділяються за рівнями. До форм високого рівня відносяться дерева виводу. До форм середнього рівня відносяться послідовні записи (наприклад, префіксні, в яких оператор передує операндам операції, або постфіксні, в яких спершу ідуть операнди), тріади та тетради. Форми низького рівня складаються з наборів команд, що практично в точності відповідають цільовій мові, оскільки вони повинні легко перетворюватися на цільовий код (прикладами є CIL для C# [2] або байт-код Java [3]).

В трансляторах найчастіше використовуються форми внутрішнього подання кількох рівнів, форми високого рівня можуть будуватися неявно, як, наприклад, при синтаксичному аналізі методом рекурсивного спуску. Як правило, компілятори мають синтаксичний блок, що переводить вихідну програму в форму середнього рівня, яка має бути зручною для машинонезалежної оптимізації. Розглянемо спершу послідовні форми.

В послідовних формах подання оператори розташовуються в тому порядку, в якому вони повинні виконуватися. Це полегшує подальшу генерацію об'єктного коду [4; 5; 6; 7]. Значного поширення серед таких форм набув польський інверсний запис (ПОЛІЗ).

1.1. Польський інверсний запис (ПОЛІЗ)

Розглянемо польський запис на прикладі арифметичних виразів. В ході трансляції арифметичний вираз має переглядатися зліва направо, і по мірі перегляду повинна генеруватися послідовність дій з використанням проміжних ре-

зультатів. Наприклад, вираз **a+(b*c-d)** має бути переведений в послідовність наступних команд:

R1: =a;

R2 : =b;

R3 : =c;

R2 : =R2*R3;

R3: =d;

R2: =R2 – R3;

R1: =R1+R2.

Ці команди можна розділити на два типи:

1) команди, що заповнюють комірки проміжних результатів;
2) команди, що виконують деяку операцію над вмістом двох останніх комірок проміжних результатів.

Наведену групу команд можна скоротити, видаливши всі посилання до комірок проміжних результатів і переписавши її як послідовність

a, b, c, *, d, -, +.

Дана послідовність може бути використана як команди, що визначають порядок обчислень з використанням стека (або магазинної пам'яті). Будь-який символ, що є операндом, заноситься в стек, а по символу оператора виконується дія над верхніми комірками стека, результат заноситься на місце використаних операндів.

Такий підхід покладений в основу використання **польського інверсного запису (ПОЛІЗ)** – бездужкового запису, розробленого австралійським філософом Чарльзом Хемблiном на основі польської нотації Яна Лукашевіча (польського математика) [8].

Звичний запис арифметичних виразів називається **інфіксним**, польський інверсний (або польський зворотний) запис називається **постфіксним**.

У постфіксному записі знак операції слідує відразу за її операндами. Таким чином, **a+b** записується як **ab+.**

Формально постфіксний запис деякого виразу **E** можна визначити через перелік правил [5]:

1) якщо **E –** змінна або константа, то постфіксний запис для **E** буде **ПОЛІЗ(E) = E**;
2) якщо **E** – це вираз типу **E1 op E2**, де **op** – бінарний оператор, то **ПОЛІЗ(E) = ПОЛІЗ(E1) ПОЛІЗ(E2) op**;
3) якщо **E** – вираз в дужках **(E1)**, то **ПОЛІЗ(E) = ПОЛІЗ(E1)**.

Наведемо приклади переведення інфіксних записів арифметичних виразів до ПОЛІЗ.

Приклад 1.1

Інфіксний запис	**Польський інверсний запис**
(a + b) * c	a b + c *
a + b * c	a b c * +
a * b + c	a b * c +

Існують різні модифікації ПОЛІЗу й алгоритми переведення інфіксного запису в ПОЛІЗ і назад. Але для всіх алгоритмів та модифікацій діє наступне правило.

> Одному арифметичному виразу в інфіксному записі відповідає один і тільки один вираз в ПОЛІЗ.

Наприклад, вираз **a+b*(c+d)*(e+f),** перекладений у ПОЛІЗ, буде таким: **abcd+*ef+*+**, не зважаючи на алгоритм, за яким було отримано цей ПОЛІЗ.

Сформулюємо основні правила, що стосуються польського запису.

> 1. Ідентифікатори в ПОЛІЗ знаходяться в тому ж порядку, що й в інфіксному записі.

2. Оператори в ПОЛІЗ розміщені в тому порядку, в якому вони повинні виконуватися (зліва направо, при чому порядок може бути відмінним від початкового).
3. Оператори розташовуються безпосередньо за своїми операндами.

Таким чином, можна записати наступні синтаксичні правила для ПОЛІЗу бінарних операцій:

<операнд>::= i |<операнд><операнд><оператор>

<оператор>::= + | – | * | / |.

Одномісний мінус можна обробляти двома способами: або записувати як бінарний оператор, тобто замість **-b** писати **0-b**, або ввести для унарного мінуса новий символ, наприклад **@,** і використовувати ще одне синтаксичне правило

<операнд>→<операнд>@

З використанням @ запис **a+(-b+c*d)** запишеться як **a b @ c d * + +**.

1.1.1. Обчислення ПОЛІЗу арифметичних виразів

Сформулюємо правила обчислення ПОЛІЗ з використанням стека.

1. Якщо поточний символ ПОЛІЗу є ідентифікатором або константою, то його значення заноситься в стек і виконується перехід до наступного символу.
2. Якщо поточний символ ПОЛІЗу є двомісним оператором, то він виконується над двома верхніми комірками стека, результат заноситься в другу (нижчу) комірку, а перша, тобто голова стека, видаляється.
3. Якщо поточний символ – одномісний оператор, то він виконується над вмістом верхньої комірки стека, який замінюється отриманим результатом.

Приклад 1.2

Проілюструємо (Таблиця 1.1) обчислення, сформованого раніше, виразу **a b @ c d * + +**, що відповідає вихідному виразу **(a+(-b+c*d))** (значення в комірках стеку відокремлені вертикальними рисками).

Таблиця 1.1. Обчислення виразу **ab@cd*++**

№ кроку	**Стек**	**ПОЛІЗ**
1.		ab@cd*++
2.	a	b@cd*++
3.	a | b	@cd*++
4.	a | -b	cd*++
5.	a | -b | c	d*++
6.	a | -b | c | d	*++
7.	a | -b | c*d	++
8.	a | -b+c*d	+
9.	a+(-b+c*d)	

1.1.2. Побудова ПОЛІЗ при висхідному розборі

Для переведення інфіксного запису в формат ПОЛІЗу можуть бути використані будь-які алгоритми граматичного розбору. Крім того, граматики можуть включати вказівки щодо перекладу коду в ПОЛІЗ, іноді такі граматики називають «транслюючими» [9; 10; 11]. При побудові ПОЛІЗ на основі висхідного розбору визначним є те, що розбір ведеться зліва направо – кожного разу редукується найлівіша проста фраза [12]. ПОЛІЗ будується одночасно з редукцією, тобто під час заміни основи нетерміналом. З того, що розбір ведеться зліва направо, робимо наступний висновок.

Якщо в основі зустрівся нетермінал, то відповідну цьому нетерміналу частину ланцюжка ПОЛІЗу вже згенеровано

Вважатимемо, що ланцюжок ПОЛІЗу зберігається в одновимірному масиві **P** (елемент масиву **P[k]** містить один символ ПОЛІЗу), і що ми маємо доступ до стека **S** (елемент **S[j]**), який використовується під час висхідного розбору [1 сс. 130-132].

Розглянемо побудову ПОЛІЗ на прикладі арифметичного виразу, в якому припускається використання одномісного мінуса. Будемо використовувати наступну граматику простого передування для арифметичних виразів, побудовану в першій частині посібника [1 сс. 137-139].

E1::=E

E::=E+T1 | E-T1 | T1 | -T1

T1::=T

T::=T*F | T/F | F

F::=(E1) | i

Кожному правилу граматики простого передування поставимо у відповідність семантичну підпрограму, тобто перелік дій, що будуть виконуватися під час редукції за цим правилом з метою формування відповідної частини ПОЛІЗу.

Семантичні підпрограми

Які ж дії повинні включати семантичні підпрограми? Розглянемо правило **E→E+T1**.

До моменту редукції основи **E+T1** підланцюжки, що відповідають **E** та **T1**, вже було редуковано, значить, в ПОЛІЗ уже згенеровано фразу

... **ПОЛІЗ(E) ПОЛІЗ(T1)**

Оскільки розбір ведеться зліва направо, правіше від **ПОЛІЗ(T1)** проміжний код ще не сформовано. Таким чином, все що потрібно від семантичної підпрограми – це занести «+» у ПОЛІЗ. Значить семантична підпрограма повинна мати вигляд

P[k]=«+»; k=k+1;

Це означає, що в поточний **k**-ий елемент ПОЛІЗу записується «+», і номер поточного елемента збільшується на одиницю, тобто виконується перехід на наступний елемент.

Аналогічно визначаються семантичні підпрограми для правил:

E → E-T1

T → T*F

T → T/F

Розглянемо підпрограму для правила **F→i,** де **i –** довільний ідентифікатор або константа. В момент редукції за цим правилом в голові стека знаходиться **i.** Оскільки в ПОЛІЗ операнди розміщуються перед відповідними операторами і зберігають такий самий порядок послідовності, що і в початковому інфіксному записі, то завдання семантичної підпрограми перенести **i** в масив **P**, зчитавши його зі стека (елемент **S[j]**):

P[k]=S[j]; k=k+1.

Семантична підпрограма, пов'язана з правилом **F→(E1)**, нічого не виконує, оскільки в ПОЛІЗі дужок немає, а для **E1** польський запис уже побудовано. Семантичні підпрограми інших правил також порожні.

В результаті для правил граматики арифметичного виразу отримаємо наступні семантичні підпрограми (Таблиця 1.2).

Таблиця 1.2. Семантичні підпрограми для граматики простого передування арифметичного виразу

№	Правила	Семантичні підпрограми
1.	E1 → E	Немає
2.	E → E+T1	P[k]= +; k=k+1
3.	E → E-T1	P[k]= -; k=k+1
4.	E → T1	Немає
5.	E → -T1	P[k]= @; k=k+1
6.	T1 → T	Немає
7.	T → T*F	P[k]= *; k=k+1

Продовження таблиці 1.2

№	Правила	Семантичні підпрограми
8.	T → T/F	P[k]= /; k=k+1
9.	T → F	Немає
10.	F → (E1)	Немає
11.	F → i	P[k]= S[j]; k=k+1

Визначимо[1] відношення передування (Таблиця 1.3) між елементами об’єднаного словника наведеної граматики.

Таблиця 1.3. Відношення передування граматики арифметичного виразу з унарним мінусом

	E1	E	T1	T	F	i	(	)	+	-	*	/	#
E1								≐					·>
E								·>	≐	≐			·>
T1								·>	·>	·>			·>
T								·>	·>	·>	≐	≐	·>
F								·>	·>	·>	·>	·>	·>
i								·>	·>	·>	·>	·>	·>
(	≐	<·	<·	<·	<·	<·	<·			<·			·>
)								·>	·>	·>	·>	·>	·>
+			≐	<·	<·	<·	<·						·>
-			≐	<·	<·	<·	<·						·>
*					≐	<·	<·						·>
/					≐	<·	<·						·>
#	<·	<·	<·	<·	<·	<·	<·	<·	<·	<·	<·	<·	

[1] Процес визначення відношень передування детально описано в попередній частині посібника [1 сс. 127-139]

На основі наведених відношень передування, за допомогою визначених семантичних підпрограм спробуємо побудувати ПОЛІЗ для арифметичного виразу **A*(-B+C)**, в процесі виконання висхідного синтаксичного розбору (Таблиця 1.4).

Таблиця 1.4. Покроковий висхідний аналіз та побудова ПОЛІЗ фрази **A*(-B+C)**

Крок	**Стек**	**Відношення**	**Вхідний ланцюжок**	**ПОЛІЗ**
1.	#	⋖	A*(-B+C)#	
2.	# A ⋖	⋗	*(-B+C)#	
3.	# F ⋖	⋗	*(-B+C)#	A
4.	# T ⋖	≐	*(-B+C)#	A
5.	# T * ⋖ ≐	⋖	(-B+C)#	A
6.	# T * (⋖ ≐ ⋖	⋖	-B+C)#	A
7.	# T * (- ⋖ ≐ ⋖ ⋖	⋖	B+C)#	A
8.	# T * (- B ⋖ ≐ ⋖ ⋖ ⋖	⋗	+C)#	A
9.	# T * (- F ⋖ ≐ ⋖ ⋖ ⋖	⋗	+C)#	A B
10.	# T * (- T ⋖ ≐ ⋖ ⋖ ⋖	⋗	+C)#	A B
11.	# T * (- T1 ⋖ ≐ ⋖ ⋖ ≐	⋗	+C)#	A B
12.	# T * (E ⋖ ≐ ⋖ ⋖	≐	+C)#	A B @
13.	# T * (E + ⋖ ≐ ⋖ ⋖ ≐	⋖	C)#	A B @
14.	# T * (E + C ⋖ ≐ ⋖ ⋖ ≐ ⋖	⋗	)#	A B @
15.	# T * (E + F ⋖ ≐ ⋖ ⋖ ≐ ⋖	⋗	)#	A B @ C

Продовження табл. 1.4

Крок	Стек	Відношення	Вхідний ланцюжок	ПОЛІЗ
16.	# T * (E + T ⋖ ≐ ⋖ ⋖ ≐ ⋖	⋗	)#	A B @ C
17.	# T * (E + T1 ⋖ ≐ ⋖ ⋖ ≐ ≐	⋗	)#	A B @ C
18.	# T * (E ⋖ ≐ ⋖ ⋖	⋗	)#	A B @ C +
19.	# T * (E1 ⋖ ≐ ⋖ ≐	≐	)#	A B @ C +
20.	# T * (E1) ⋖ ≐ ⋖ ≐ ≐	⋗	#	A B @ C +
21.	# T * F ⋖ ≐ ≐	⋗	#	A B @ C +
22.	# T ⋖	⋗	#	A B @ C + *
23.	# T1 ⋖	⋗	#	A B @ C + *
24.	# E ⋖	⋗	#	A B @ C + *
25.	# E1 ⋖	⋗	#	A B @ C + *

Тут показано основний підхід до опису синтаксису та семантики «транслюючої» граматики, хоча в даному випадку багато семантичних підпрограм не виконують жодних дій. Польський інверсний запис отримується як побічний продукт під час синтаксичного аналізу. Розглянутий спосіб перекладу арифметичних виразів в ПОЛІЗ базується на граматиці, в якій непрямим чином враховано послідовність операцій. Наприклад, операція «*» редукується раніше, ніж «+». Якщо граматика не враховує послідовність операцій, цей підхід складно використовувати.

1.1.3. Алгоритм побудови ПОЛІЗу Дейкстри

ПОЛІЗу притаманні дві важливі властивості:

1) описувані ним дії можна виконати (або програмувати) в процесі одностороннього перегляду зліва направо без повернень, оскільки в момент

появи знаку операції відомі місцезнаходження й обчислені значення відповідних операндів;

2) операнди розташовані в тому ж порядку, як і в початковому записі, тому переклад в ПОЛІЗ зводиться до зміни порядку знаків операцій.

Саме ці властивості дають підстави використовувати ПОЛІЗ як проміжну мову подання програм. З першої властивості виходить легкий переклад ПОЛІЗу в машинні команди. З другої – проста побудова ПОЛІЗ за допомогою стека та пріоритетів операцій.

Для побудови ПОЛІЗ можуть використовуватися різні алгоритми [13] (вище було розглянуто алгоритм переведення ПОЛІЗ при висхідному розборі). Найбільшого поширення набув алгоритм, запропонований нідерландським ученим у галузі комп'ютерних наук Едсгером Дейкстрою (1930-2002). Автор дав цьому алгоритму назву «сортувальна станція», оскільки він нагадує операції, що виконуються на залізничних сортувальних станціях.

Алгоритм Дейкстри [14] оснований на використанні пріоритетів виконуваних операцій. Наприклад, для арифметичних виразів операціям можна надати наступні пріоритети (Таблиця 1.5).

Для пояснення методу Дейкстри використовується аналогія із залізничною станцією, що має форму Т-подібного роз'їзду (Рисунок 1.1). В якості гілки для виконання маневру або переупорядковування виступає стек (магазин).

Алгоритм Дейкстри

1. Ідентифікатори та константи проходять від входу прямо до виходу. Операції звичайно потрапляють на вихід через магазин.
2. Якщо пріоритет операції, що знаходиться в стеку, не менший за пріоритет поточної вхідної операції, то операція зі стека подається на вихід і п.2 повторюється, інакше поточна операція заноситься в стек.
3. Якщо стек порожній, то поточна операція заноситься в стек.
4. Якщо вхідний ланцюжок порожній, всі операції зі стека по черзі передаються на вихід за ознакою кінця виразу (наприклад, «**;**»).

Таблиця 1.5. Пріоритети арифметичних операцій

Операція	**Пріоритет**
+, -	0
*, /	1
^ [2]	2

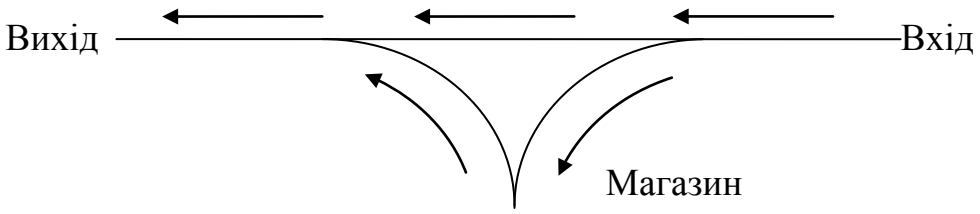


Рисунок 1.1 – Т-подібний роз'їзд залізничної станції

Приклад 1.3

Побудуємо ПОЛІЗ виразу **a+b*c/d** за алгоритмом Дейкстри.

Крок 1. На вході першим знаходиться ідентифікатор **a**, за першим пунктом алгоритму він одразу передається на вихід.

Крок 2. На вході знаходиться оператор «+», діємо за третім пунктом алгоритму і записуємо її в магазин.

Крок 3. Ідентифікатор **b** передається з входу на вихід, і «сортувальна станція» набуває наступного вигляду (Рисунок 1.2).

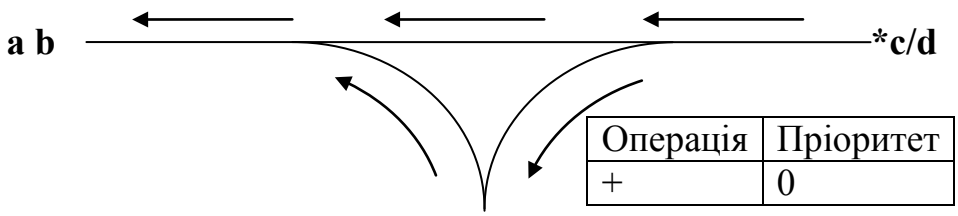
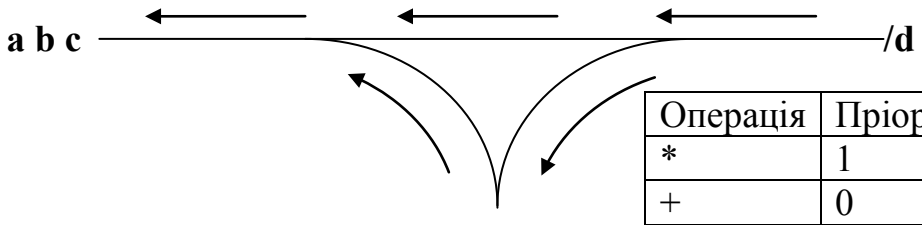


Рисунок 1.2

Крок 4. Першим на вході стоїть оператор «*». Діємо за другим пунктом алгоритму: оскільки пріоритет операції «*» вище за пріоритет «+», знак «*» заноситься в стек.

[2] ^ позначає операцію піднесення до ступеня

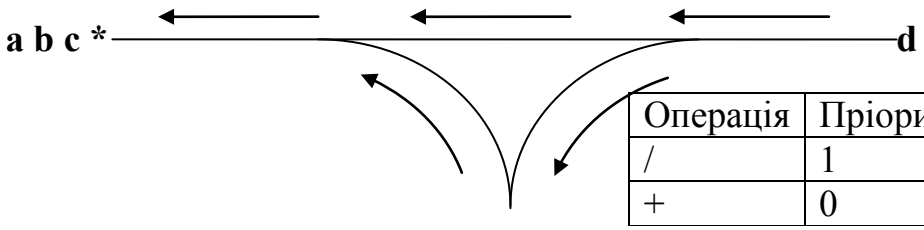
Крок 5. Ідентифікатор **c** передається з входу на вихід, отриманий стан «сортувальної машини» відображено нижче (Рисунок 1.3).



Операція	Пріоритет
*	1
+	0

Рисунок 1.3

Крок 6. Оскільки пріоритет операції «/» дорівнює пріоритету «*», діємо за другим пунктом алгоритму: «*» виштовхується на вихід. Повторюємо другий пункт алгоритму: порівнюємо пріоритети вхідного символу («/») та вершини стека («+»). Оскільки пріоритет «+» менший за пріоритет «/», знак «/» заноситься в стек, отримаємо наступну картину (Рисунок 1.4).



Операція	Пріоритет
/	1
+	0

Рисунок 1.4

Крок 7. Змінна **d** передається на вихід.

Крок 8. Оскільки вхідний ланцюжок вичерпано, діємо за четвертим пунктом алгоритму: передаємо на вихід вміст стеку. Отримуємо **a b c * d / +.**

Особливості обробки унарного мінуса

Операція зміни знаку повинна виконуватися раніше двомісного «+» або «–», тобто повинна мати пріоритет, як у «*» або «/». Особливістю унарного мінуса є те, що в початковій програмі він записується так само, як і бінарний, проте для формування ПОЛІЗ його треба відрізняти від бінарного мінуса. Ці два різновиди мінусів можна розрізняти за місцеположенням в початковій програмі. Наприклад, згідно до третього правила наведеної вище граматики ариф-

метичного виразу (Таблиця 1.2) бінарний мінус може стояти в початковій програмі лише після терміналів, що належать множині **Last$^+$(E) = {T1, T, F,), i}**, тобто після «**)**» або «**i**». Таким чином, для визначення, чи є поточний мінус бінарним, необхідно перевірити попередню лексему: якщо вона є «**)**» або «**i**», то даний мінус є бінарним, в іншому разі – унарним.

Особливості обробки дужок

Деякі особливості при формуванні ПОЛІЗу мають лексеми-обмежувачі, такі як дужки: «**(**» і «**)**». Як було сказано раніше, ПОЛІЗ не містить дужок, проте розташування дужок в інфіксному записі вказує на порядок виконання інших операцій. Таким чином, дужки повинні впливати на побудову ПОЛІЗу. Для того, щоб операції між дужками потрапляли в ПОЛІЗ у першу чергу, відкриваюча дужка «**(**» повинна мати найнижчий пріоритет, але записуватися в стек, нічого не виштовхуючи. Закриваюча дужка «**)**» повинна мати на одиницю більший пріоритет, щоб виштовхувати все до «**(**» (але не далі) і видаляти «**(**». Причому «**(**» ніколи не передається на вихід, а «**)**» не заноситься в стек. Усі інші операції повинні мати пріоритет не менший за пріоритет «**)**» (Таблиця 1.6).

Таблиця 1.6. Таблиця пріоритетів арифметичних виразів з дужками

Операція	**Пріоритет**
(	0
), +, **-**	1
@, *, /	2
^	3

Приклад 1.4

Використовуючи цей ряд пріоритетів (Таблиця 1.6), побудуємо ПОЛІЗ виразу **- a + b * c ^ (d / e) / f** за алгоритмом Дейкстри, хід побудови будемо відображати у вигляді таблиці (Таблиця 1.7).

Таблиця 1.7. Побудова ПОЛІЗ(**- a + b * c ^ (d / e) / f**)

ПОЛІЗ		a	@	b		c			d		e	/	^	f	/
													*		+
Стек										/	/				
								(	(	(	(				
							^	^	^	^	^	^			
					*	*	*	*	*	*	*	*	/	/	
	@	@	+	+	+	+	+	+	+	+	+	+	+	+	
Вхідний символ ланцюжка	-	a	+	b	*	c	^	(	d	/	e	)	/	f	

Результуючий ПОЛІЗ: **a @ b c d e / ^ * f / +**

Розширення ПОЛІЗ на логічні вирази

Розглянемо граматику логічних виразів.

<ЛВ> ::= <ЛТ> | <ЛВ> or <ЛТ>

<ЛТ> ::= <ЛМ> | <ЛТ> and <ЛМ>

<ЛМ> ::= <відн.> | not <ЛМ> | [<ЛТ>]

<відн.>::=<вираз> <знак відн.> <вираз>

<знак відн.> ::= ≠ | ≤ | ≥ | < | > | =

<вираз> ::= <терм.> | <вираз> + <терм.>| <вираз>- <терм.> | – <терм.>

<терм.> ::= <множ> | <терм.> * <множ> | <терм.> / <множ>

<множ> ::= <перв.в.> | <множ> ^ <перв.в.>

<перв.в.> ::= (<вираз>) | id | con

Щоб перекладати в ПОЛІЗ логічні вирази за алгоритмом Дейкстри, необхідно визначити пріоритети для операцій відношення та логічних операцій.

Операції відношення. Справа і зліва від операції відношення знаходяться арифметичні вирази. Значить, пріоритет операції відношення має бути таким, щоб виштовхувати з магазина всі операції, що відносяться до арифметич-

ного виразу, який передує операції відношення. Для цього пріоритет операцій відношення має бути меншим за пріоритет будь-якої арифметичної операції.

Аналогічно **логічні операції** повинні виштовхувати зі стека операції відношення та арифметичні операції, які їм передують. Таким чином, пріоритети логічних операцій мають бути меншими за пріоритет операцій відношення. Також слід враховувати, що в першу чергу серед логічних операцій виконується логічне заперечення (**not**), потім операція кон'юнкції (**and**), і в останню чергу – операція диз'юнкції (**or**) (Таблиця 1.8).

Таблиця 1.8. Пріоритети для граматики логічних виразів

Операція	**Пріоритет**
([	0
)] or	1
And	2
not	3

Операція	**Пріоритет**
< > = <> ≤ ≥	4
+ –	5
@ * /	6
^	7

Приклад 1.5

Побудуємо ПОЛІЗ для логічного виразу **not a + b > 3 and [c = 0 or (a – q) = 1]**, хід побудови будемо відображати у вигляді таблиці (Таблиця 1.9).

Таблиця 1.9. Побудова ПОЛІЗ «not a+b >3 and [c=0 or (a-q)=1]»

ПОЛІЗ		a		b	+	3	>		c		0	=		a		q	-		1	=	and
							not													or	
Стек															-	-					
													(	(	(	(		=	=		
										=	=	or	or	or	or	or	or	or	or		
			+	+	>	>		[	[	[	[	[	[	[	[	[	[	[	[		
	not	not	not	not	not	not	and	and	and	and	and	and	and	and	and	and	and	and	and	and	
Вх. символ	not	a	+	b	>	3	and	[	c	=	0	or	(	a	-	q	)	=	1	]	

Результуючий ПОЛІЗ:

a b + 3 > not c 0 = a q – 1 = or and.

Польський запис дуже просто розширити. Треба тільки дотримуватися правила, що за операндами повинен слідувати оператор.

ПОЛІЗ для оператора присвоєння

Оператор присвоєння:

<змінна> ≔ <вираз>

у польському записі повинний мати вигляд:

<змінна> <вираз> ≔

Оператор присвоєння **≔** виконується інакше ніж операція відношення «дорівнює» =. Після його виконання і **<змінна>**, і **<вираз>** мають бути виключені зі стека, оскільки в даній грамаиці оператор **≔**, не має результуючого значення, що заноситься в стек, на відміну від двомісних операцій, які зустрічаються у виразах та відношеннях. Крім того, в стеку має використовуватися не значення лексеми **<змінна>**, а її адреса, оскільки значення **<вираз>** має запам'ятовуватися за адресою **<змінна>**. Ці особливості необхідно враховувати при формуванні та виконанні польського запису.

Отже, оператор присвоєння **a ≔ b + c** в ПОЛІЗ виглядатиме так: **abc+≔.**

З цього витікають вимоги до величини пріоритету оператора присвоєння.

По-перше, пріоритет **≔** має бути менше пріоритету знаку будь-якої арифметичної операції, оскільки оператор присвоєння виконується після обчислення виразу, записаного праворуч від нього. Якщо в граматиці допускається присвоєння змінним результатів логічних виразів, пріоритет знаку оператора присвоєння має бути меншим і за пріоритети знаків усіх логічних операцій та операцій відношення.

По-друге, пріоритет знаку **≔** має бути більше пріоритету знаків кінця оператора (**;**, **end**, **else**, тощо), щоб знак кінця оператора виштовхував **≔** на вихід.

Якщо в мові оператор присвоєння має результат, і допускається вираз

a≔b≔5,

то за наявності одного пріоритету для ≔ оператор транслюватиметься в

ab≔5≔

при цьому обидві команди не можуть бути виконані правильно. На момент зчитування другого знаку присвоєння з ПОЛІЗу вже буде видалено запис **ab≔**, тобто оператор не матиме першого операнда. Тому для символу вводять два пріоритети: порівняльний та стековий. Стековий пріоритет має оператор ≔, якщо він знаходиться в стеку, а порівняльний – оператор ≔ з вхідного ланцюжка. Порівняльний пріоритет слід обрати більшим за пріоритет будь-якого іншого знаку, щоб ≔ не виштовхував зі стека інших знаків (Таблиця 1.10). Тоді наведений вираз **a≔b≔5;** транслюватиметься в

a b 5 ≔≔.

Таблиця 1.10. Пріоритети для граматики оператора присвоєння

Знак	**Пріоритет**	
	Стековий	**Порівняльний**
([	0	
)]	1	
≔	2	10
or	3	
and	4	
not	5	
< ≤ > ≥ = <>	6	
+ -	7	
* / @	8	
^	9	

Згенеровані в ПОЛІЗ знаки «:=» повинні оброблятися по різному. Після виконання першого в стеку має залишитися значення, що було присвоєно (в даному випадку **5**) цільовій змінній (**b**), як у випадку виконання арифметичних операцій. Натомість після виконання присвоєння за другим знаком «:=» і змінна, і значення мають бути видалені зі стека. Отже слід розрізняти ці два знаки, тому в ПОЛІЗ повинні генеруватися дві різні команди. Позначатимемо команду присвоєння з результуючим значенням **Пр**, тоді

ПОЛІЗ(a:=b:=5) = a b 5 Пр :=

Виконаємо отриманий ПОЛІЗ, хід виконання відображатимемо в табличному вигляді (Таблиця 1.11).

Таблиця 1.11. Хід виконання ПОЛІЗ **a b 5 Пр :=**

№ кроку	Стек	ПОЛІЗ	Примітки
1.		a b 5 Пр :=	
2.	a	b 5 Пр :=	
3.	a | b	5 Пр :=	
4.	a | b | 5	Пр :=	За адресою **b** заносимо **5**
5.	a | 5	:=	За адресою **a** заносимо **5**
6.			

Приклад 1.6

Використовуючи наведені пріоритети (Таблиця 1.10), побудуємо ПОЛІЗ для наступного ланцюжка:

F := P := not [(2 + k) * i < 0 and H ≥145]

Хід побудови будемо відображати у вигляді таблиці (Таблиця 1.12), для наочності пріоритет оператора присвоєння позначатимемо у верхньому індексі.

Таблиця 1.12. Побудова ПОЛІЗ(**F ≔ P ≔ not [(2 + k) * i < 0 and H ≥145]**)

ПОЛІЗ	F		P					2		k	+		i	*	0	<	H		145	≥	not
																				and	Пр
																					≔
Стек									+	+								≥	≥		
							(	(	(	(		*	*	<	<	and	and	and	and		
						[	[	[	[	[	[	[	[	[	[	[	[	[	[		
					not	not	not	not	not	not	not	not	not	not	not	not	not	not	not	not	
				:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	
		:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	:=[2]	
Вх. символ	F	:=[10]	P	:=[10]	not	[	(	2	+	k	)	*	i	<	0	and	H	≥	145	]	

Результуючий ПОЛІЗ:

F P 2 k + i * 0 < H 145 ≥ and not Пр ≔

1.1.4. Переклад в ПОЛІЗ операторів умовного переходу

До цих пір розглядалися вирази і оператори, в яких порядок дій визначається тільки пріоритетами операцій і дужками, і в ході виконання програми не змінюється.

Проте в мовах звичайно є умовні переходи, цикли й інші оператори з динамічно змінним порядком операцій.

Для того, щоб записати послідовність дій, які породжуються такими операторами, необхідно ввести в ПОЛІЗ додаткові операції виконання переходів і мітки, що позначатимуть, куди слід перейти.

Введемо наступні операції: **БП**[3] – безумовний перехід і **УПХ**[4] – умовний перехід по хибності.

Команда **БП** повинна мати один операнд, що позначатиме місце в програмі, на яке слід перейти. Таким чином, синтаксис цієї команди буде наступним:

[3] Команда БП аналогічна команді асемблера JMP
[4] Команда УПХ схожа на команду асемблера JNE

m БП,

де **m** – мітка, на яку здійснюється безумовний перехід.

Команда **УПХ** повинна мати два операнди, один з них позначатиме цільове місце переходу, а інший – результат логічного виразу, визначимо її синтаксис таким чином:

ПЛВ m УПХ,

де **ПЛВ** – ПОЛІЗ логічного виразу;

m – мітка, на яку виконується перехід.

Якщо значення **ПЛВ** – хиба, то здійснюється перехід на мітку **m**. Якщо – істина, то виконується оператор, наступний за **УПХ**.

За допомогою введених команд можна будувати ПОЛІЗ для різноманітних умовних переходів та циклів.

Розглянемо оператор умовного переходу наступного вигляду:

if <ЛВ> then <опер1> else <опер2> ; **(1.1)**

Для розуміння послідовності дій, що викликає оператор (1.1), побудуємо блок-схему (Рисунок 1.5) його виконання.

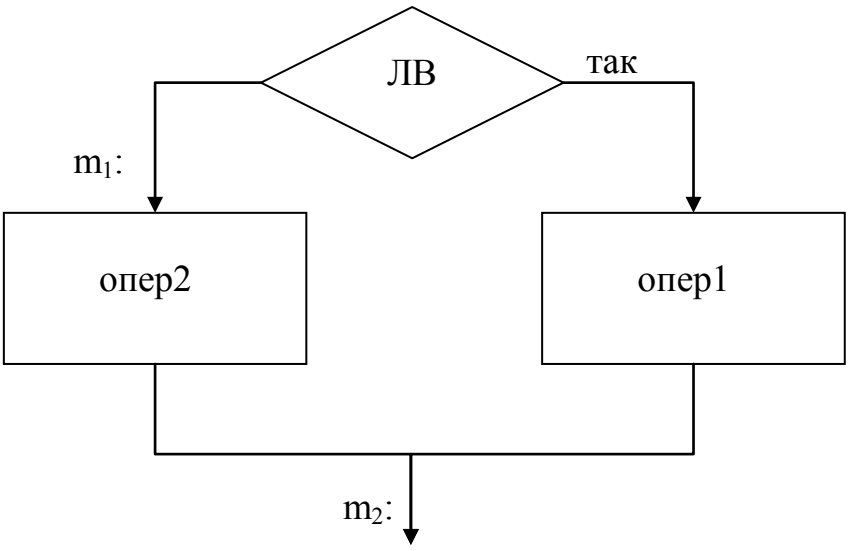


Рисунок 1.5 – Блок-схема оператора розгалуження

Для побудови ПОЛІЗ оператора (1.1) будемо використовувати введену раніше команду **УПХ**. Позначимо (Рисунок 1.5) міткою $\mathbf{m_1}$ перехід по хибі, а міткою $\mathbf{m_2}$ вихід з оператора. Опишемо цю блок-схему ПОЛІЗом:

ПЛВ m_1 УПХ Попер1 m_2 БП m_1: Попер2 m_2:

де **ПЛВ** = ПОЛІЗ(<ЛВ>),

Попер1 = ПОЛІЗ(<опер1>),

Попер2 = ПОЛІЗ(<опер2>).

Порівняємо отриманий ПОЛІЗ з початковим записом оператора (Рисунок 1.6) і визначимо додаткове навантаження на службові слова вхідного оператора, а також їхні пріоритети.

Початковий вираз	if	<ЛВ>	then	<опер1>	else	<опер2>	;
Вихідний ПОЛІЗ		ПЛВ	m_1 УПХ	Попер1	m_2 БП m_1:	Попер2	m_2:

Рисунок 1.6 – ПОЛІЗ умовного оператора

Очевидно, по **then** треба генерувати в ПОЛІЗ фрагмент: $\mathbf{m_1}$ **УПХ**, по **else** – фрагмент $\mathbf{m_2}$ **БП** $\mathbf{m_1}$**:**, по символу **;** – фрагмент $\mathbf{m_2}$**:**.

Визначимо особливості алгоритму перекладу в ПОЛІЗ оператора умовного переходу.

Окрім того, що обмежувачі **if**, **then**, **else**, **;** генерують в ПОЛІЗ специфічні фрагменти, вони грають в деякому розумінні роль дужок.

Символ **if** подібний до **(**, а символи **then** і **else** відіграють роль закриваючих дужок для попереднього виразу і відкриваючих – для наступного. Тому символу **if** задається пріоритет **0**, а **then** та **else** – пріоритет **1**.

Особливості обробки окремих символів.

1. Символ **if**, що має пріоритет **0**, як і будь-яка відкриваюча дужка, записується в стек. Цей символ використовується як «зберігач» і «переносник» робочих міток для операцій **УПХ** і **БП**. При появі символу **then**, до запису **if** додається перша мітка $\mathbf{m_i}$, а після появи символу

else до нього додається друга мітка $\mathbf{m_{i+1}}$, тут **i** – номер чергової робочої мітки.

2. Символ **then** з пріоритетом **1** виштовхує зі стека всі знаки до першого **if** виключно. При цьому генерується нова мітка $\mathbf{m_i}$ й у вихідний рядок заноситься $\mathbf{m_i}$ **УПХ**. Потім мітка $\mathbf{m_i}$ заноситься в таблицю міток і дописується у вершину стека, таким чином, запис **if** у вершині стека перетворюється на запис **if** $\mathbf{m_i}$.
3. Символ **else** з пріоритетом **1** виштовхує зі стека всі знаки до першого **if** виключно. У вершині стека знаходиться запис **if** $\mathbf{m_i}$. У вихідний рядок додається запис $\mathbf{m_{i+1}}$ **БП** $\mathbf{m_i}$**:**. Потім мітка $\mathbf{m_{i+1}}$ заноситься в таблицю міток і дописується у вершину стека до запису **if** $\mathbf{m_i}$, отримуємо в стеку **if** $\mathbf{m_i}$ $\mathbf{m_{i+1}}$.
4. Символ кінця умовного оператора (наприклад, **;** або **end**) виштовхує зі стека все до найближчого **if**. Це може бути **if** $\mathbf{m_i}$ $\mathbf{m_{i+1}}$ (або **if** $\mathbf{m_i}$ у разі конструкції **if <вираз> then <оператор>;**). Запис видаляється зі стека, а у вихідний рядок додається $\mathbf{m_{i+1}}$**:** (або $\mathbf{m_i}$**:** у випадку неповного оператора умовного переходу). При розборі вкладених умовних операторів (**if <вир1> then <опер1> else if <вир2> then <опер2> else if…**) в стеку може бути послідовність:

$$\mathbf{if\ m_i\ m_{i+1}}$$

$$\mathbf{if\ m_{i-2}\ m_{i-1}}$$

$$\mathbf{\dots}$$

$$\mathbf{if\ m_{i-2k}\ m_{i-2k+1}}$$

По символу кінця оператора вся ця послідовність видаляється зі стека, і по кожному запису у вихідний рядок додається остання мітка. Таким чином, у вихідний рядок буде занесено запис вигляду:

$$\mathbf{m_{i+1}\ :\ m_{i-1}:\ \dots\ m_{i-2k+1}\ :}$$

Подамо описані особливості у вигляді таблиці (Таблиця 1.13).

Таблиця 1.13. Особливості обробки **if**, **then**, **else**

Символ	**Запис у вершину стеку**	**Запис у вихідний рядок**
if	**if**	**-**
then	**if m_i**	**m_i УПХ**
else	**if m_i m_{i+1}**	**m_{i+1} БП m_i:**
Кінець оператора	Очищення стека, в тому числі видалення записів типу **if m_i m_{i+1}** зі стека	Останні мітки з кожного запису **if** зі стека **m_{i+1} : m_{i-1}: … m_{i-2k+1} :**

Приклад 1.7

Виконаємо трансляцію в ПОЛІЗ наступного оператора:

if a>b then a:=5 else a:=0;

Процес трансляції відображатимемо покроково (Таблиця 1.14).

Таблиця 1.14. Приклад трансляції в ПОЛІЗ умовного оператора

Вихід		a		b	> m_1 УПХ	a		5	:= m_2 БП m_1 :	a		0	:= m_2 :
Стек	if	if	> if	> if	if m_1	if m_1	:= if m_1	:= if m_1	if m_1m_2	if m_1m_2	:= if m_1m_2	:= if m_1m_2	
Вх. рядок	if	a	>	b	then	a	:=	5	else	a	:=	0	;

Результуючий ПОЛІЗ:

a	**b**	**>**	**m_1**	**УПХ**	**a**	**5**	**:=**	**m_2**	**БП**	**m_1 :**	**a**	**0**	**:=**	**m_2 :**
1	2	3	4	5	6	7	8	9	10		11	12	13	14

У коді наявна операція «**:**» (**m_1:**, **m_2:**), виконання якої полягає в наступному: вона встановлює значення мітки, тобто заносить в таблицю міток адресу чергової робочої команди, яка помічена цією міткою. Сама операція «**:**», та її операнд (мітка) може бути видалена з оброблюваного рядка ПОЛІЗу, оскільки вона не описує дії вихідної програми.

У даному випадку таблиця міток матиме наступний вигляд (Таблиця 1.15).

Таблиця 1.15. Таблиця міток для прикладу 1.7

Ім'я мітки	**Значення**
m_1	11
m_2	14

Розглянемо приклад з вкладеним оператором умовного переходу.

Приклад 1.8

Початковий текст оператора:

if a>b then a:=5 else if a<b then a:=10 else a:=0;

Опишемо процес трансляції (Таблиця 1.16).

Таблиця 1.16. Переклад в ПОЛІЗ вкладеного умовного оператора

Вихід		a		B	> m_1 УПХ	a		5	:= m_2 БП m_1:		a
Стек	if		> if		if m_1		:= if m_1		if m_1 m_2	if if m_1 m_2	
Вх. рядок	if	a	>	B	then	a	:=	5	else	if	a

Продовження таблиці 1.16

Вихід		b	< m_3 УПХ	a		10	:= m_4 БП m_3:	a		0	:= m_4: m_2:
Стек	< if if m_1 m_2		if m_3 if m_1 m_2		:= if m_3 if m_1 m_2		if m_3 m_4 if m_1 m_2		:= if m_3 m_4 if m_1 m_2		
Вх. рядок	<	b	then	a	:=	10	else	a	:=	0	;

Результуючий ПОЛІЗ:

a b > m_1 УПХ a 5 := m_2 БП m_1: a b < m_3 УПХ a 10 := m_4 БП m_3: a 0 := m_4: m_2:

Розглянемо приклади використання складених операторів замість <опер1> і <опер2> в операторі умовного переходу (1.1).

if <ЛВ> then <список опер1> else <список опер2> ;

Приклад 1.9

Вхідний текст оператора:

if a = b then a:= 0 ; b := 1 else a := 1 ;

Опишемо процес трансляції (Таблиця 1.17).

Тут складений оператор після **then** не поміщений в операторні дужки тому необхідно забезпечити правильну роботу першого символу **;** після оператора присвоєння **a := 0**. Коли надходить перший символ **;** , у стеку знаходиться **if**, і, відповідно до визначених раніше особливостей обробки, символ **;** повинен оброблятися як символ закінчення оператора **if**, вивантажуючи стек і генеруючи відповідний фрагмент на вихід. Але перший **;** не є ознакою кінця оператора, він просто розділяє оператори списку. Щоб правильно обробляти

даний символ, треба аналізувати додаткові ознаки, наприклад, що в стеку при **if** тільки одна мітка.

Таблиця 1.17. Переклад в ПОЛІЗ умовного оператора зі складеним оператором після **then** без операторних дужок

Вихід		a		b	= m_1 УПХ	a		0	≔	b		1	:= m_2 БП m_1 :	A		1	:= m_2 :
Стек	if		= if		if m_1		:= if m_1		if m_1		:= if m_1		if m_1 m_2		:= if m_1 m_2		
Вх. рядок	if	a	=	b	then	a	:=	0	;	b	:=	1	else	a	:=	1	;

Результуючий ПОЛІЗ:

a b = m_1 УПХ a 0 ≔ b 1 ≔ m_2 БП m_1: a 1 ≔ m_2:

Описана проблема розрізнення кінцевого та розділового знаку **;** не виникає при використанні операторних дужок (наприклад, **begin** та **end**). Механізм обробки операторних дужок аналогічний обробці дужок арифметичного та логічного виразів.

Приклад 1.10

Опишемо процес трансляції (Таблиця 1.18) оператора:

if a=b then begin a:=0; goto M end else a:=1;

У даному прикладі використовується команда **goto**, яка при перенесенні зі стека в ПОЛІЗ перетворюється на команду **БП**. Пріоритет **goto** має бути більшим за пріоритет відкриваючих операторних дужок, щоб не виштовхувати їх зі стека. Також команда **goto** повинна виштовхуватися закриваючою операторною дужкою, тому її пріоритет має бути не менше пріоритету закриваючої операторної дужки (тобто не менше одиниці). Призначимо команді **goto** пріоритет **1**.

Таблиця 1.18. Переклад в ПОЛІЗ умовного оператора зі складеним оператором після **then** поміщеним операторні дужки

Вихід		a		b	= m_1 УПХ		a		0	:=		M	БП	m_2 БП m_1 :	a		1	:= m_2 :
Стек	if		= if		if m_1	begin if m_1		:= begin if m_1		begin if m_1	goto begin if m_1		if m_1	if $m_1 m_2$		:= if $m_1 m_2$		
Вх. рядок	if	a	=	b	then	begin	a	:=	0	;	goto	M	end	else	a	:=	1	;

Результуючий ПОЛІЗ:

a b = m_1 УПХ a 0 ≔ M БП m_2 БП m_1 : a 1 ≔ m_2 :

Приклад 1.11

Розглянемо приклад трансляції в ПОЛІЗ (Таблиця 1.19) умовного оператора зі складеним оператором після **else**. Вхідний текст оператора:

if a < b then a := c else begin a := 2; k := 1 end

Таблиця 1.19. Переклад в ПОЛІЗ умовного оператора зі складеним оператором після **else**, поміщеним в операторні дужки

Вихід		a		b	< m_1 УПХ	a		c	:= m_2 БП m_1 :		a		2	:=	k		1	:-
Стек	if		< if		if m_1		:= if m_1		if m_1 m_2	begin if $m_1 m_2$		:= begin if $m_1 m_2$		begin if $m_1 m_2$		:= begin if $m_1 m_2$		if $m_1 m_2$
Вх. рядок	if	a	<	b	then	a	≔	c	else	begin	a	≔	2	;	k	≔	1	end

Результуючий ПОЛІЗ:

a b < m_1 УПХ a c ≔ m_2 БП m_1 : a 2 ≔ k 1 ≔m_2 :

Часто в комп'ютерних програмах використовуються скорочені оператори розгалуження. Такі оператори не описують дії по не виконанню умови розгалуження, тобто в них відсутня секція **else**. Найчастіше умовою виступає перевірка наявності деякої інформації або ввімкненої ознаки.

Визначимо правила побудови ПОЛІЗ скороченого оператора умовного переходу типу

if <ЛВ> then <опер1>;

Побудуємо блок-схему (Рисунок 1.7) виконання оператора розгалуження без секції **else**.

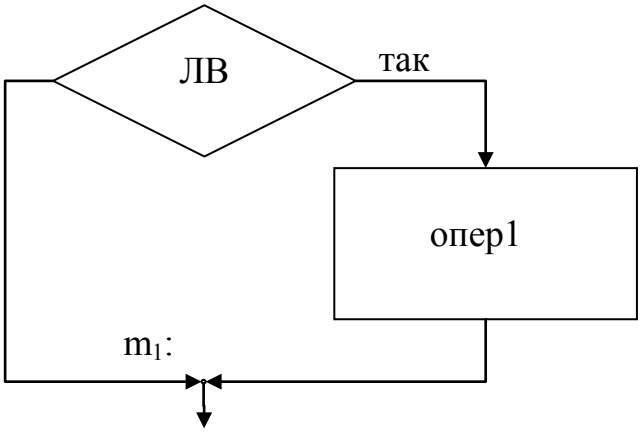


Рисунок 1.7 – Блок-схема скороченого оператора умовного переходу

За наведеною блок-схемою побудуємо ПОЛІЗ скороченого оператора умовного переходу (Рисунок 1.8).

Початковий вираз	if	<ЛВ>	then	<опер1>	;
Вихідний ПОЛІЗ		ПЛВ	m_1 УПХ	Попер1	m_1:

Рисунок 1.8 – ПОЛІЗ скороченого оператора розгалуження

Як видно, при порівнянні запису скороченого оператора умовного переходу в загальному вигляді з вихідним ПОЛІЗом (Рисунок 1.8), навантаження на

символи **if**, **then** та **;** таке саме, як і в повному операторі умовного переходу. Оскільки в операторі відсутня гілка **else**, для виконання описаних дій достатньо використовувати лише одну робочу мітку. Таким чином, символ **;** генерує в ПОЛІЗ мітку $\mathbf{m_1}$**:**, яка є водночас і першою, і останньою міткою при **if**, що знаходиться в стеку.

Приклад 1.12

Розглянемо приклад побудови ПОЛІЗ скороченого оператора розгалуження (Таблиця 1.20).

Початковий текст оператора:

if a>b then a:=0;

Таблиця 1.20. Побудова ПОЛІЗ скороченого оператора розгалуження

Вихід		a		b	> m_1 УПХ	a		0	:= m_1 :
Стек	if		> if		if m_1		:= if m_1		
Вх. рядок	if	a	>	b	then	a	:=	0	;

Результуючий ПОЛІЗ:

a b > $\mathbf{m_1}$ УПХ a 0 ≔ $\mathbf{m_1}$:

У деяких мовах програмування зустрічаються оператори вигляду:

if <ЛВ> then goto M;

де **M** – мітка, описана в початковому програмному коді користувача.

Очевидно, описаний оператор є окремим випадком попереднього скороченого оператора розгалуження (Рисунок 1.9), тільки замість виклику довільної інструкції (<опер1>) в тілі розгалуження викликається команда **goto**. Фактично, такий оператор дозволяє описати повноцінне розгалуження: по справдженню умови

(<ЛВ>) виконуються дії, описані після визначеної користувачем мітки **M**; а по хибності – дії, що знаходяться після даного умовного оператора але перед **M**.

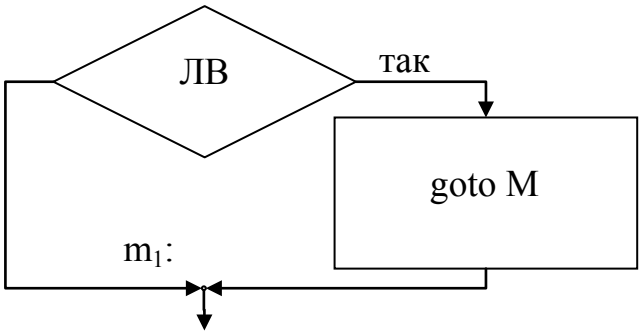


Рисунок 1.9 – Блок-схема оператора розгалуження з командою goto

За наведеною схемою отримуємо наступний ПОЛІЗ (Рисунок 1.10).

Початковий вираз	if	<ЛВ>	then	goto M	;
Вихідний ПОЛІЗ		ПЛВ	m_1 УПХ	M БП	m_1:

Рисунок 1.10 – Навантаження на службові слова оператора розгалуження з викликом команди goto

Як видно, навантаження на символи **if**, **then** та **;** не змінюється.

Приклад 1.13

Побудуємо (Таблиця 1.21) **ПОЛІЗ(if a>b then goto M;)**.

Таблиця 1.21. Побудова ПОЛІЗ оператора розгалуження з **goto**

Вихід		a		b	> m_1 УПХ		M	БП m_1 :
Стек	if		> if		if m_1	goto if m_1		
Вх. рядок	if	a	>	b	then	goto	M	;

Зустрічаються подібні мовні конструкції, але без службового слова **then**, тобто оператор має вигляд:

if <ЛВ> goto M;

В цьому випадку організацію умовного переходу (а саме, формування команди **УПХ**) можна покласти на **goto** за умови, що в стеку **if**. Отже крім звичайного навантаження команда **goto** буде мати додаткове. Блок-схема даного оператора співпадає з блок-схемою попередньо описаного (Рисунок 1.9).

Розглянемо приклад реалізації (Рисунок 1.11).

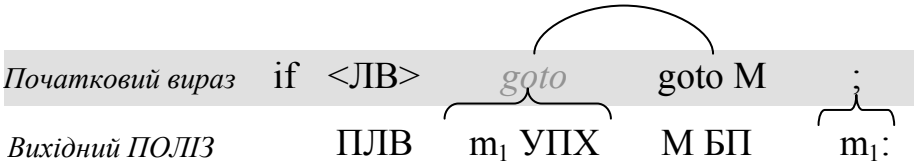


Рисунок 1.11 – Навантаження на службові слова для **if <ЛВ> goto M;**

В даному випадку команда **goto** виштовхує зі стеку всі оператори, відповідно до свого пріоритету. Додатково, якщо в стеку знаходиться **if m_1**, в ПОЛІЗ генерується запис **m_1 УПХ**. Після чого **goto** працює відповідно до свого пріоритету, тобто записується в стек і при виштовхуванні генерує в ПОЛІЗ запис **БП**.

Приклад 1.14

Розглянемо (Таблиця 1.22) конкретний приклад побудови ПОЛІЗ для оператора умовного переходу з **goto**.

Початковий текст оператора: **if a>b goto M;**.

Таблиця 1.22. Побудова **ПОЛІЗ(if a>b goto M;)**

Вихід		a		b	> m_1 УПХ	M	БП m_1 :
Стек	if		> if		goto if m_1		
Вх. рядок	if	a	>	b	goto	M	;

Результуючий ПОЛІЗ: **a b > m_1 УПХ M БП m_1 :**

Реалізацію останнього оператора можна дещо модернізувати. Відповідно до цього оператора перехід на мітку **M** виконується, якщо логічний вираз – істина. Команда ж **УПХ** працює по хибності. Отже достатньо згенерувати заперечення логічного виразу (**<ЛВ>**), і мітка $\mathbf{m_1}$ взагалі не знадобиться.

Відповідно до вищесказаного наведемо інший варіант реалізації оператора вигляду

if <ЛВ> goto M;

Його блок-схема є окремим випадком блок-схеми попереднього оператора (Рисунок 1.9), побудуємо її наступним чином (Рисунок 1.12).

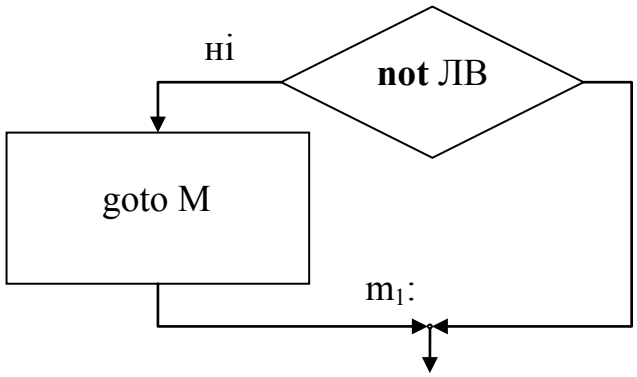


Рисунок 1.12 – Модифікована блок-схема оператора умовного переходу на мітку

ПОЛІЗ буде мати наступний вигляд:

ПЛВ not M УПХ

Наведений ПОЛІЗ можна отримати за умови наступного навантаження на службові слова:

- по **if** в стек записується оператор заперечення **not**, після символу **if**;
- команда **goto** виштовхує зі стека все, включаючи **if**, і записується в стек;
- символ **«;»**, виштовхуючи **goto** зі стека, замінює його на **УПХ**.

Приклад 1.15

Розглянемо приклад побудови ПОЛІЗ (Таблиця 1.23) для оператора умовного переходу на мітку без використання робочих міток.

Початковий текст оператора: **if a>b goto M;**.

Таблиця 1.23. Побудова **ПОЛІЗ(if a>b goto M;)** без робочих міток

Вихід		a		b	> not	M	УПХ
Стек	not if		> not if		goto		
Вх. рядок	if	a	>	b	goto	M	;

Результуючий ПОЛІЗ: **a b > not M УПХ**

Перевагою даної модифікації, безперечно, є економія ресурсів, що виділяються на робочі мітки. Проте він має і недоліки. По-перше, обробка зарезервованого слова **if** набуває додаткових особливостей, що може бути шкідливим у випадку наявності в мові програмування кількох видів операторів умовного переходу. По-друге, обробка команди **goto** не відповідає її змісту, що може спричинити додаткові труднощі, якщо в мові є окрема команда **goto <мітка>**.

1.1.5. Переклад в ПОЛІЗ операторів циклу

Існують різноманітні конструкції операторів циклу, які можна звести до двох основних типів:

- тип перелічення

for k:=a while b do A ;

- тип арифметичної прогресії

for k:=a step h to c do A ;

де **k** – ім'я змінної параметра циклу;

b – логічний вираз;

a, h, c – арифметичні вирази, що визначають початкове значення параметра циклу, крок зміни параметра циклу та його кінцеве значення, відповідно;

A – тіло циклу.

Для трансляції в ПОЛІЗ будемо використовувати команди умовного і безумовного переходу **УПХ** і **БП**. Для розробки алгоритму перекладу конкретного виду оператора спочатку будемо формувати його блок-схему, на основі якої визначатимемо загальний вигляд ПОЛІЗу. Порівнюючи початковий загальний вигляд оператора з його ПОЛІЗом, визначатимемо додаткове навантаження на службові слова. Правильність алгоритму будемо перевіряти на конкретному прикладі перекладу.

Простий цикл перелічення

Розглянемо оператор циклу перелічення наступного виду

for k ≔ a while b do A;

Побудуємо блок-схему (Рисунок 1.13) роботи цього оператора.

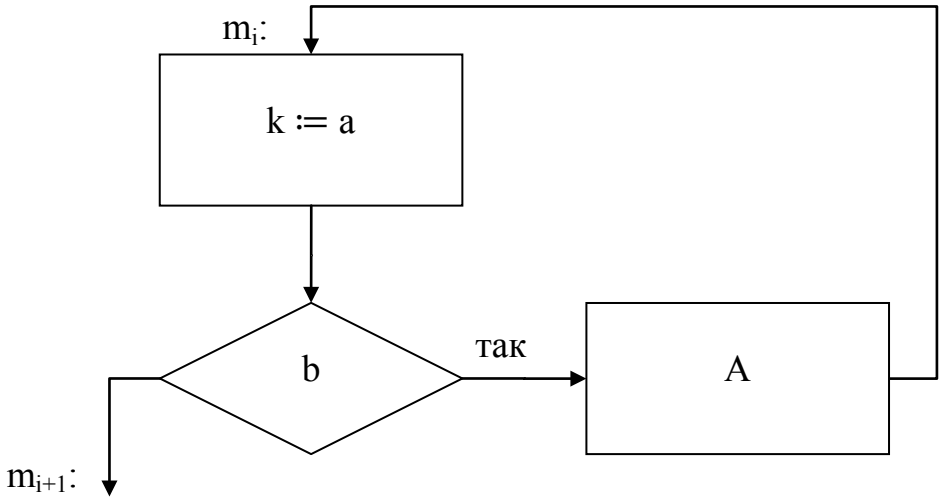


Рисунок 1.13 – Блок-схема простого циклу перелічення

Очевидно, для реалізації будуть потрібні дві робочі мітки $\mathbf{m_i}$ та $\mathbf{m_{i+1}}$.

Запишемо реалізацію отриманої блок-схеми в ПОЛІЗ і порівняємо її з початковим загальним виглядом оператора (Рисунок 1.14), якщо ПА = ПОЛІЗ(А).

Початковий вираз	for	k := a	while	b	do	A	;		
Вихідний ПОЛІЗ	m_i :	k a :=		b	m_{i+1} УПХ	ПА	m_i БП m_{i+1}:		

Рисунок 1.14 – Навантаження на службові слова для простого циклу

Порівнюючи отриманий код з початковим текстом оператора (Рисунок 1.14), визначаємо додаткове навантаження на службові слова та їхні пріоритети:

- **for** має пріоритет **0**; генерує чергову робочу мітку $\mathbf{m_i}$, заносить в стек запис **for** $\mathbf{m_i}$, заносить в ПОЛІЗ $\mathbf{m_i}$ **:** ; символ **for** грає роль зберігача та переносника робочих міток, як і символ **if** в операторах умовного переходу;
- **while** – пріоритет 1, генерує чергову робочу мітку $\mathbf{m_{i+1}}$, змінює в стеку запис **for** $\mathbf{m_i}$ на **for** $\mathbf{m_i}$ $\mathbf{m_{i+1}}$, в ПОЛІЗ нічого додатково не заносить;
- **do** – пріоритет 1, заносить в ПОЛІЗ $\mathbf{m_{i+1}}$ **УПХ**.
- **;** – пріоритет 1, заносить в ПОЛІЗ $\mathbf{m_i}$ **БП** $\mathbf{m_{i+1}}$**:** і видаляє запис **for** зі стека.

Всі службові слова та знак **;** діють спочатку відповідно до своїх пріоритетів, виштовхуючи зі стека все до запису **for** $\mathbf{m_i}$ $\mathbf{m_{i+1}}$ виключно. А потім, використовуючи мітки, вказані при **for**, генерують в ПОЛІЗ відповідні фрагменти.

Приклад 1.16

Виконаємо трансляцію простого циклу перелічення в ПОЛІЗ, етапи побудови ПОЛІЗу подаватимемо у вигляді таблиці (Таблиця 1.24).

Початковий текст оператора:

for k := a while c > d do c := c+1;

Результуючий ПОЛІЗ:

m_1 : k a := c d > m_2 УПХ c c l + := m_1 БП m_2 :

Таблиця 1.24. Переклад в ПОЛІЗ простого оператора циклу

Вихід		l		a	:=	c		c	> m_2 УПХ	c		c		l	+ := m_1 БП m_2 :
Стек	for m_1		:= for m_1		for m_1m_2		> for m_1m_2		for m_1m_2		:= for m_1m_2		+ := for m_1m_2		
Вх. р.	for	k	:=	a	while	c	>	d	do	c	:=	c	+	l	;

Розглянемо інші види операторів циклу.

Цикл перелічення з передумовою

Загальний вигляд оператора:

while b do A ;

Побудуємо блок-схему циклу перелічення з передумовою (Рисунок 1.15).

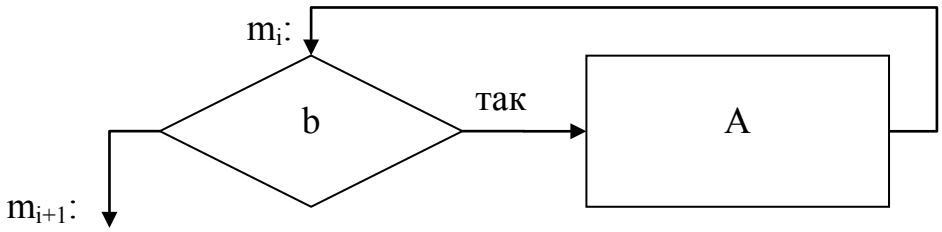


Рисунок 1.15 – Блок-схема циклу з передумовою

Використовуючи ті самі команди умовного та безумовного переходу **УПХ** і **БП**, запишемо цей алгоритм в ПОЛІЗ (Рисунок 1.16) і визначимо навантаження на службові слова.

Початковий вираз	while	b	do	A	;
Вихідний ПОЛІЗ	m_i :	b	m_{i+1} УПХ	ПА	m_i БП m_{i+1}:

Рисунок 1.16 – Навантаження на службові слова для циклу з передумовою

Порівнюючи отриманий текст з початковим, визначимо пріоритети та функціональне навантаження службових слів і знаків.

- **while** має пріоритет 0; генерує робочу мітку $\mathbf{m_i}$; записує в стек **while** $\mathbf{m_i}$; генерує в ПОЛІЗ $\mathbf{m_i}$ **:** . Виконує роль зберігача та переносника міток в стеку.
- **do** – пріоритет 1; відповідно до свого пріоритету, очищує стек до **while** виключно; якщо знаходить в стеку **while**, генерує робочу мітку $\mathbf{m_{i+1}}$ і заносить в ПОЛІЗ $\mathbf{m_{i+1}}$ **УПХ**.
- **;** – пріоритет 1; відповідно до свого пріоритету, очищує стек до **while** виключно; якщо знаходить в стеку **while**, генерує в ПОЛІЗ $\mathbf{m_i}$ **БП** $\mathbf{m_{i+1}}$**:** і видаляє запис **while** зі стека.

Приклад 1.17

Виконаємо трансляцію в ПОЛІЗ описаного вище оператора циклу з передумовою: **while a > b do a := a – k;**

Хід трансляції оформимо у вигляді таблиці (Таблиця 1.25).

Таблиця 1.25. Переклад в ПОЛІЗ циклу з передумовою

Вихід	m_1 :	a		b	> m_2 УПХ	a		a		k	- := m_1 БП m_2 :
Стек	while m_1		> while m_1		while m_1		:= while m_1		- := while m_1		
Вх. рядок	while	a	>	b	do	a	:=	a	-	k	;

Результуючий ПОЛІЗ: $\mathbf{m_1}$ **: a b >** $\mathbf{m_2}$ **УПХ a a k – :=** $\mathbf{m_1}$ **БП** $\mathbf{m_2}$ **:**

Цикл перелічення з постумовою

Загальний вигляд оператора:

repeat A until b;

де **A** – тіло циклу;

b – умова, що може бути логічним виразом або відношенням.

Блок-схема оператора наступна (Рисунок 1.17).

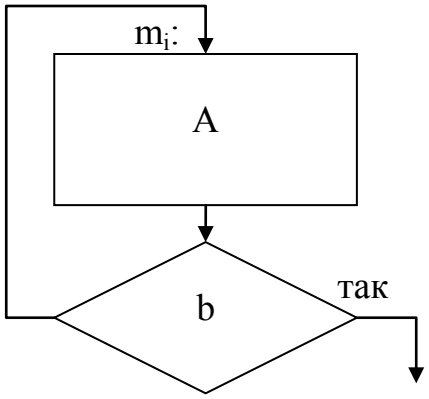


Рисунок 1.17 – Блок-схема циклу з постумовою

Запишемо цей алгоритм у формі ПОЛІЗ (Рисунок 1.18), покладемо **Пb = ПОЛІЗ(b)**, **ПА = ПОЛІЗ(А)**, як і раніше.

Початковий вираз	repeat	A	until	b	;
Вихідний ПОЛІЗ	m_i :	ПА		Пb	m_i УПХ

Рисунок 1.18 – Навантаження на службові слова для циклу з постумовою

Функціональне навантаження та пріоритети службових слів і символів:

- **repeat** має пріоритет 0; генерує робочу мітку $\mathbf{m_i}$; записує в стек **repeat** $\mathbf{m_i}$; генерує в ПОЛІЗ $\mathbf{m_i}$ **:** ;
- **until** – пріоритет 1; за пріоритетом виштовхує все зі стеку на вихід до **repeat** виключно;
- **;** – пріоритет 1; працює за пріоритетом (очищає стек до **repeat**, виключно); якщо в стеку зустрічається **repeat**, генерує в ПОЛІЗ $\mathbf{m_i}$ **УПХ** і видаляє запис **repeat** зі стека.

Приклад 1.18

Відповідно до вищесказаного, виконаємо трансляцію (Таблиця 1.26) в ПОЛІЗ деякого циклу з постумовою.

Початковий текст оператора:

repeat a ≔ a +1 until a > b;

Таблиця 1.26. Переклад в ПОЛІЗ циклу з постумовою

Вихід	m_1 :	a		a		1	+ ≔	a		b	> m_1 УПХ
Стек	repeat m_1		≔ repeat m_1		+ ≔ repeat m_1		repeat m_1		> repeat m_1		
Вх. рядок	repeat	a	≔	a	+	1	until	a	>	b	;

Результуючий ПОЛІЗ:

m_1 : a a 1 + := a b > m_1 УПХ

Оператор циклу типу арифметичної прогресії з правильним порядком параметрів

Загальний вигляд оператора:

for k := a step h to c do A ;

де **a, h, c** – арифметичні вирази, що позначають початкове значення параметра циклу, крок зміни параметра циклу та кінцеве значення параметра циклу, їм відповідають ПОЛІЗи **Па, Пh, Пс,** відповідно;

k – ім'я ідентифікатора циклу;

A – тіло циклу, відповідний йому ПОЛІЗ – **ПА**.

Побудуємо блок-схему виконання цього оператора (Рисунок 1.19).

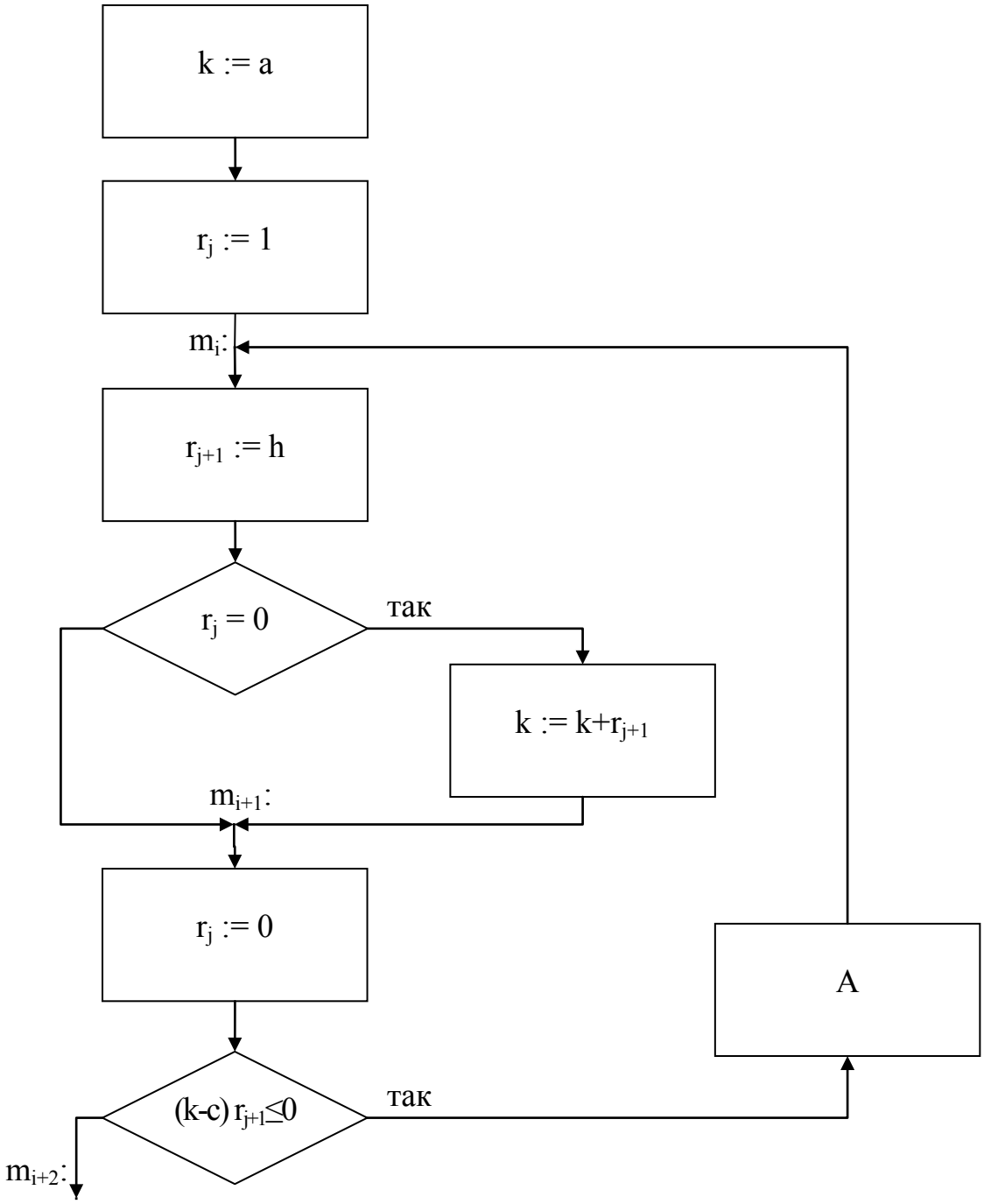


Рисунок 1.19 – Блок-схема циклу арифметичної прогресії

Окрім робочих міток $\mathbf{m_i}$, $\mathbf{m_{i+1}}$, $\mathbf{m_{i+2}}$ для трансляції знадобилися дві додаткові робочі комірки $\mathbf{r_j}$ та $\mathbf{r_{j+1}}$. Комірка $\mathbf{r_j}$ потрібна для збереження ознаки того, що тіло циклу виконується вперше, і значення параметра циклу **k** не потрібно збільшувати. Під час першого виконання циклу $\mathbf{r_j}$ скидається в **0**, тому під час наступних проходів циклу значення параметра **k** збільшується на визначений користувачем крок **h**.

Значення кроку **h** використовується в алгоритмі двічі: при модифікації значення параметра циклу **k** та при перевірці умови виходу з циклу. Через те, що **h** може виражатися достатньо складним арифметичним виразом, результуюче значення якого може відрізнятися для різних ітерацій циклу, для повторного використання **h** не можна використовувати просто змінну, як у випадку параметра циклу **k**. Тому для зберігання значення кроку **h** необхідно ввести додаткову робочу комірку – $\mathbf{r_{j+1}}$.

Кінцеве значення параметра циклу **c** використовується в алгоритмі один раз при формуванні умови виходу з циклу. Під час побудови ПОЛІЗ треба передати на вихід ПОЛІЗ(**c**) саме у момент формування умови виходу з циклу, порядок параметрів даного циклу дозволяє це виконати без використання робочих комірок.

Ім'я параметра циклу **k** використовується кілька разів (але це просто один ідентифікатор), тому вводиться додаткова змінна параметра циклу (ЗПЦ) для зберігання імені цього ідентифікатора. Для того, щоб під час трансляції записувати в ЗПЦ правильний ідентифікатор, необхідно ввести деяку ознаку та відповідне навантаження на якийсь з операторів. Зручно записувати в ЗПЦ ідентифікатор циклу, коли він є останнім записом в ПОЛІЗі, тобто під час обробки **≔**. Введемо ознаку циклу (ОЦ) таку, що при ОЦ = 1 під час обробки **≔** останній елемент ПОЛІЗу буде записуватися в ЗПЦ.

Відповідно до блок-схеми отримаємо ПОЛІЗ (Рисунок 1.20).

Початковий вираз	for k≔a	step	h	to
Вихідний ПОЛІЗ	k Па ≔	r_j1≔m_i:r_{j+1}	Пh	≔r_j0=m_{i+1} УПХ kkr_{j+1}+≔m_{i+1}:r_j0≔k

Продовження

Початковий вираз	c	do	A	;
Вихідний ПОЛІЗ	Пc	-r_{j+1}*0 ≤ m_{i+2} УПХ	ПА	m_i БП m_{i+2}:

Рисунок 1.20 – Навантаження на службові слова для циклу арифметичної прогресії

Порівнюючи отриманий код з початковим текстом оператора, визначаємо додаткові функції службових слів та їхні пріоритети при побудові ПОЛІЗ:

- **for** грає роль відкриваючої дужки, тому має пріоритет 0; генерує три робочі мітки; в стек записує **for** $\mathbf{m_i m_{i+1} m_{i+2}}$; у ПОЛІЗ нічого не заносить; ознаку циклу встановлює в 1 (ОЦ := 1);
- **:=** діє відповідно до пріоритету; а також, якщо ознака циклу дорівнює 1 (ОЦ=1), звертається до останнього елемента ПОЛІЗу (це ім'я параметра циклу) і записує його в змінну параметра циклу (ЗПЦ); після цього скидає ознаку циклу в 0 (ОЦ := 0);
- **step** є закриваючою дужкою для попереднього виразу, тому має пріоритет 1; виштовхує зі стеку все до **for** виключно; генерує робочі комірки $\mathbf{r_j}$, $\mathbf{r_{j+1}}$; записує в ПОЛІЗ $\mathbf{r_j}$ **1 :=** $\mathbf{m_i}$**:** $\mathbf{r_{j+1}}$ (мітку бере зі стеку);
- **to** так само, як і **step**, має пріоритет 1; виштовхує зі стеку все до **for**, виключно; на вихід (у ПОЛІЗ) генерує запис **:=** $\mathbf{r_j}$ **0 =** $\mathbf{m_{i+1}}$ **УПХ ЗПЦ ЗПЦ** $\mathbf{r_{j+1}}$ **+ :=** $\mathbf{m_{i+1}}$**:** $\mathbf{r_j}$ **0 := ЗПЦ** (замість ЗПЦ підставляється ім’я параметра циклу, яке зберігається в ЗПЦ, відповідно до загального вигляду даного оператора (Рисунок 1.20) в ЗПЦ знаходитиметься ім’я ідентифікатора **k**);
- **do** є закриваючою дужкою для попереднього виразу, має пріоритет 1; виштовхує зі стеку все до запису **for**, виключно; генерує на вихід запис **-** $\mathbf{r_{j+1}}$ *** 0 ≤** $\mathbf{m_{i+2}}$ **УПХ**;
- **;** (символ кінця оператора) виштовхує зі стеку все до запису **for**; на вихід генерує $\mathbf{m_i}$ **БП** $\mathbf{m_{i+2}}$**:** і видаляє запис **for** зі стеку.

Приклад 1.19

Відповідно до вищесказаного, виконаємо трансляцію (Таблиця 1.27) в ПОЛІЗ циклу арифметичної прогресії **for k := 1 step 2 to 7 do a := a + 1;** .

Результуючий ПОЛІЗ:

k 1 := $\mathbf{r_1}$ **1:=** $\mathbf{m_1}$ **:** $\mathbf{r_2}$ **2 :=** $\mathbf{r_1}$ **0 =**$\mathbf{m_2}$ **УПХ k k** $\mathbf{r_2}$ **+ :=** $\mathbf{m_2}$ **:**$\mathbf{r_1}$ **0 := k 7 –** $\mathbf{r_2}$ *** 0 ≤** $\mathbf{m_3}$ **УПХ a a 1 + :=** $\mathbf{m_1}$ **БП** $\mathbf{m_3}$ **:**

Таблиця 1.27. Переклад в ПОЛІЗ циклу арифметичної прогресії

Вихід		k		1	≔ r_1 1 := m_1: r_2	2	:= r_1 0 = m_2 УПХ k k r_2 + := m_2: r_1 0 := k	7
ОЦ	1	1	0	0	0	0	0	0
ЗПЦ			k	k	k	k	k	k
Стек	for $m_1m_2m_3$		≔ for $m_1m_2m_3$		for $m_1m_2m_3$		for $m_1m_2m_3$	
Вх. рядок	for	k	≔	1	step	2	to	7

Продовження табл. 1.27

Вихід	- r_2 * 0 ≤ m_3 УПХ	a		a		1	+ ≔ m_1 БП m_3:
ОЦ	0	0	0	0	0		
ЗПЦ	k	k	k	k	k	k	k
Стек	for $m_1m_2m_3$		≔ for $m_1m_2m_3$		+ ≔ for $m_1m_2m_3$		
Вх. рядок	do	a	≔	a	+	1	;

Оскільки після **do** робочі комірки більше не потрібні, лічильник робочих комірок (ЛРК) можна скидати по **do**, тоді для трансляції програми з будь-якою кількістю циклів (в тому числі вкладених) потрібні тільки дві робочі комірки.

Приклад 1.20

Розглянемо (Таблиця 1.28) трансляцію вкладеного оператора циклу арифметичної прогресії.

Початковий текст оператора:

for k1:=a1 step h1 to c1 do for k2:=a2 step h2 to c2 do A ;

Таблиця 1.28. Переклад в ПОЛІЗ вкладеного циклу

Вихід		**k1**		a1	≔ r_1 1 := m_1: r_2	h1	:= r_1 0 = m_2 УПХ **k1** **k1** r_2 + := m_2: r_1 0 := **k1**	c1
ОЦ	1	1	0	0	0	0	0	0
ЗПЦ			**k1**	k1	k1	k1	k1	k1
Стек	for $m_1m_2m_3$		≔ for $m_1m_2m_3$		for $m_1m_2m_3$		for $m_1m_2m_3$	
Вх. рядок	for	k1	≔	a1	step	h1	to	c1
Додаткові функції	Генерація міток m_1, m_2, m_3				Генерація робочих комірок r_1, r_2			

Продовження таблиці 1.28

Вихід	- r_2 * 0 ≤ m_3 УПХ		**k2**		a2	≔ r_1 1 := m_4: r_2	h2	:= r_1 0 = m_5 УПХ **k2** **k2** r_2 + := m_5: r_1 0 := **k2**
ОЦ	0	1	1	0	0	0	0	0
ЗПЦ	k1	k1	k1	**k2**	k2	k2	k2	k2
Стек	for $m_1m_2m_3$	for $m_4m_5m_6$ for $m_1m_2m_3$		≔ for $m_4m_5m_6$ for $m_1m_2m_3$		for $m_4m_5m_6$ for $m_1m_2m_3$		for $m_4m_5m_6$ for $m_1m_2m_3$
Вх. рядок	do	for	k2	≔	a2	step	h2	to
Додаткові функції	ЛРК ≔ 0	Генерація міток m_4, m_5, m_6				Генерація робочих комірок r_1, r_2		

Продовження таблиці 1.28

Вихід	c2	- r_2 * 0 ≤ m_6 УПХ	a		a		2	+ := m_4 БП m_6: m_1 БП m_3:
ОЦ	0	0	0	0	0	0	0	0
ЗПЦ	k2	k2	k2	k2	k2	k2	k2	k2
Стек		for $m_4m_5m_6$ for $m_1m_2m_3$		:= for $m_4m_5m_6$ for $m_1m_2m_3$		+ := for $m_4m_5m_6$ for $m_1m_2m_3$		
Вх. рядок	c2	do	a	:=	a	+	2	;
Додаткові функції		ЛРК := 0						

Результуючий ПОЛІЗ:

k1 a1 := r_1 1 := m_1 : r2 **h1** := r_1 0 = m_2 УПХ **k1 k1** r_2 + := m_2: r_1 0 := **k1 c1** – r_2 * 0 ≤ m_3 УПХ **k2 a2** := r_1 1 := m_4 : r_2 **h2** := r_1 0 = m_5 УПХ **k2 k2** r_2 + := m_5: r_1 0 := **k2 c2** – r_2 * 0 ≤ m_6 УПХ **a a 2 +** := m_4 БП m_6 : m_1 БП m_3:

Оператор циклу типу арифметичної прогресії з неправильним порядком параметрів

Розглянемо варіант оператора циклу типу арифметичної прогресії, коли кінцеве значення параметра циклу вказується раніше, ніж значення кроку.

Загальний вигляд оператора:

for k := a to c step h do A ;

Блок-схема виконання такого оператора наступна (Рисунок 1.21).

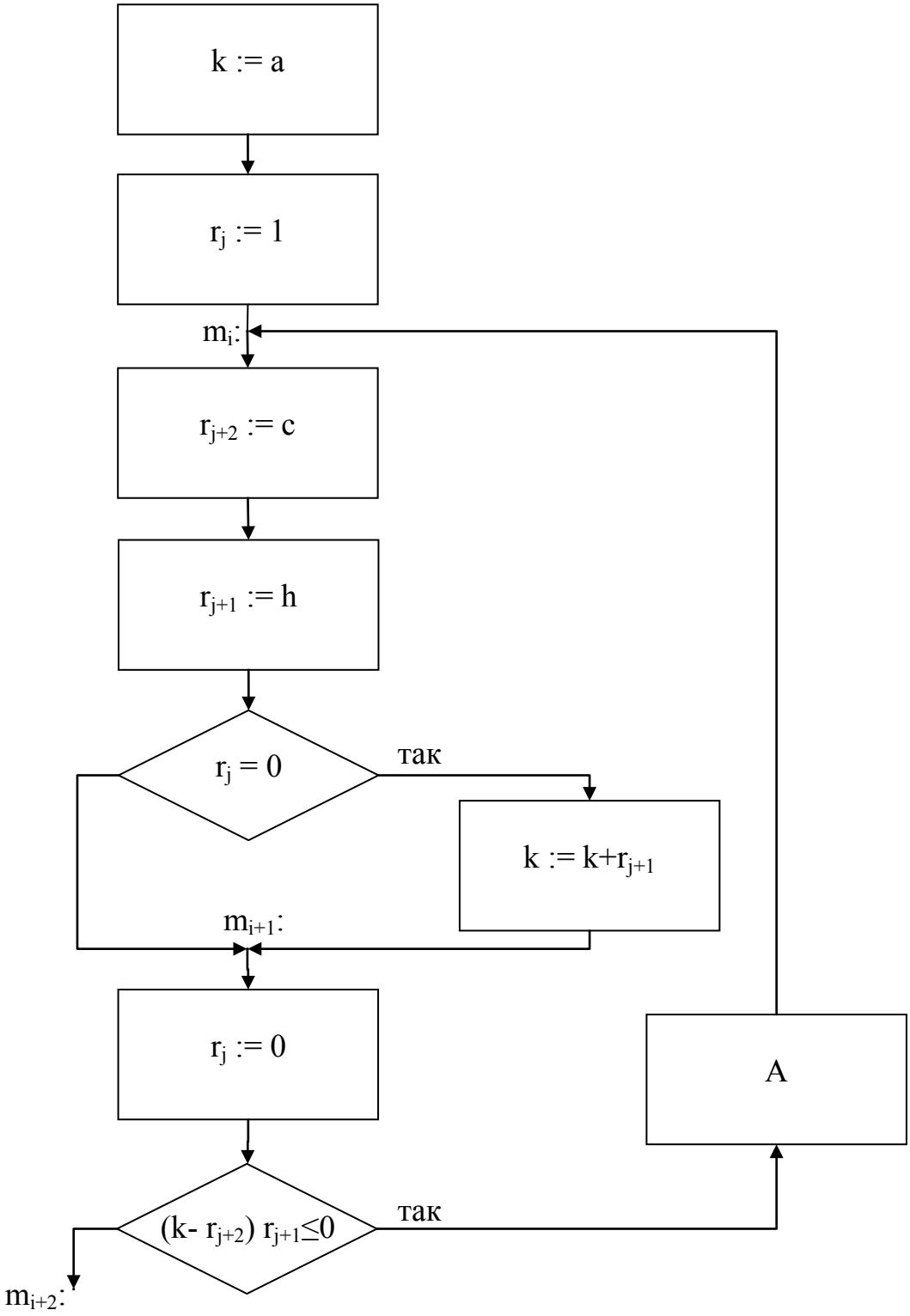


Рисунок 1.21 – Блок-схема циклу арифметичної прогресії з непрямим порядком параметрів

На відміну від попереднього варіанта циклу в загальному вигляді даного оператора кінцеве значення параметра циклу вказується перед кроком циклу, але в алгоритмі виконання циклу значення цих параметрів потребуються в іншому

порядку. Нагадуємо, що в ПОЛІЗі всі ідентифікатори зустрічаються в тому ж порядку, що і в початковому записі. Щоб задовольнити цю умову, вводиться ще одна робоча комірка $\mathbf{r_{j+2}}$, в якій зберігається кінцеве значення параметра циклу.

Побудуємо ПОЛІЗ і порівняємо його з початковим текстом, для встановлення додаткового навантаження на службові слова (Рисунок 1.22).

Початковий вираз	for	k:=a	to	c	step	h	
Вихідний ПОЛІЗ		k Па	$r_j1:=m_i:r_{j+2}$	Пс	$:=r_{j+1}$	Пh	
Продовження							
Початковий вираз			do			A	;
Вихідний ПОЛІЗ	$:=r_j0=m_{i+1}$	УПХ	k k $r_{j+1}:=m_{i+1}:r_j0:=$	k $r_{j+2}-r_{j+1}*0\leq m_{i+2}$	УПХ	ПА	m_iБПm_{i+2}:

Рисунок 1.22 – ПОЛІЗ для циклу арифметичної прогресії з непрямим порядком параметрів

При реалізації оператора циклу типу арифметичної прогресії можна виключити використання ознаки $\mathbf{r_j}$ за рахунок зміни алгоритму виконання оператора

for k := a step h until c do A ;

Оновлена блок-схема оператора буде наступною (Рисунок 1.23).

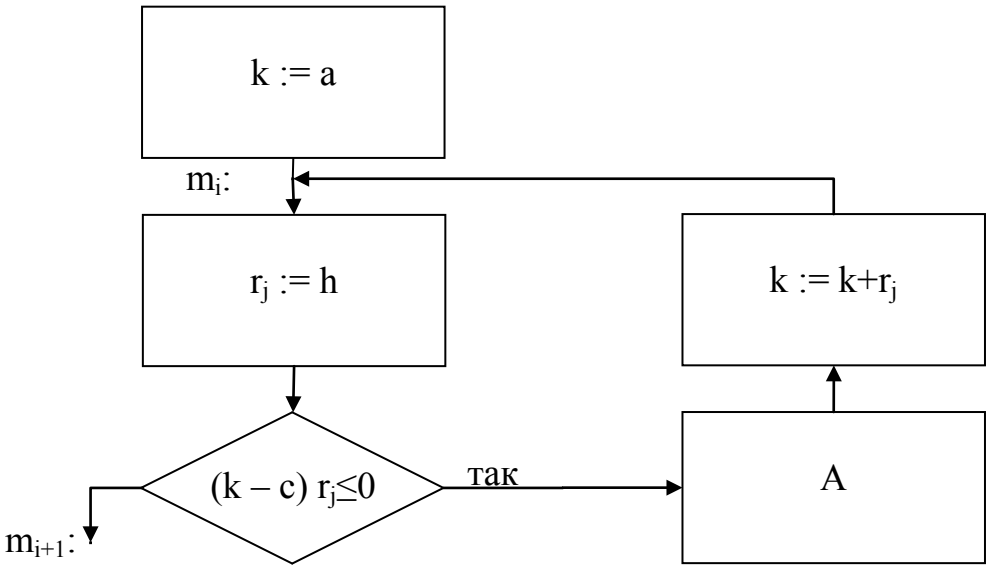


Рисунок 1.23 – Блок-схема циклу арифметичної прогресії без використання ознаки першого проходу циклу

В цьому випадку для трансляції циклу в ПОЛІЗ достатньо однієї комірки $\mathbf{r_j}$, де зберігається значення кроку параметра циклу, також зменшується кількість робочих міток, оскільки зникає одне з розгалужень алгоритму (Рисунок 1.23). Відповідно змінюються навантаження на ключові слова (Рисунок 1.24).

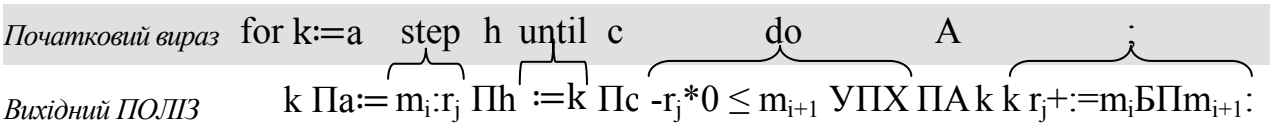


Рисунок 1.24 – Навантаження на службові слова для циклу арифметичної прогресії без використання ознаки першого проходу

Через те, що в такому випадку по **;** треба генерувати в ПОЛІЗ **k k $\mathbf{r_j}$ + := $\mathbf{m_i}$ БП $\mathbf{m_{i+1}}$:**, ім'я змінної циклу **k** і значення кроку $\mathbf{r_{j+1}}$ необхідно передавати через стек так само, як і мітки, щоб правильно перекладати в ПОЛІЗ вкладені оператори. Таким чином, не дивлячись на те, що кількість робочих міток та комірок в даному варіанті зменшується, процес трансляції ускладнюється за рахунок збільшення інформації, яку слід передавати через стек.

1.1.6. Переклад в ПОЛІЗ оператора циклу зі списком елементів

Розглянемо оператор циклу вигляду

<оператор циклу> ::= for <змінна> := <список циклу> do <оператор>

<список циклу>::= <елемент списку>| <список циклу>,<елемент списку>

Елементи списку циклу можуть бути 3-х видів:

1) типу арифметичного виразу **for k := $\mathbf{a_1}$, $\mathbf{a_2}$ ..., $\mathbf{a_n}$ do A;**
2) типу перерахунку **for k := $\mathbf{a_1}$ while $\mathbf{b_1}$, $\mathbf{a_2}$ while $\mathbf{b_2}$ do A;**
3) типу арифметичної прогресії:

for k := $\mathbf{a_1}$ step $\mathbf{h_1}$ to $\mathbf{c_1}$, $\mathbf{a_2}$ step $\mathbf{h_2}$ to $\mathbf{c_2}$ do A,

де **a**, **h**, **c** – арифметичні вирази; **b** – логічні вирази.

При створенні стратегії трансляції в ПОЛІЗ оператора циклу зі списком елементів необхідно врахувати, що:

- семантика елемента списку залежить від його типу;

- будь-який елемент списку починається арифметичним виразом і тому на початку перекладу чергового елемента його тип невідомий;
- в списку циклу різні елементи можуть зустрічатися в довільній послідовності;
- бажано, щоб алгоритм перекладу працював без повернень.

Додаткові складності обумовлені тим, що в операторах такого вигляду після виконання тіла циклу точка повернення не визначена однозначно (кожному елементу списку циклу відповідає окрема точка повернення), тому для трансляції необхідно ввести додаткові операції:

- **БПВ** – безумовний перехід з поверненням, з трьома операндами-мітками;
- **В** – повернення, що забезпечує резервування комірки ПОЛІЗу (для команди повернення, що генерується операцією БПВ).

Операція **БПВ**, у ПОЛІЗі має вигляд

$$\mathbf{m_p\ m_q\ m_r\ БПВ}$$

і означає, що необхідно:

1) виконати безумовний перехід на мітку $\mathbf{m_q}$;
2) сформувати за адресою мітки $\mathbf{m_r}$ (що зарезервована операцією **В**) команду безумовного переходу на мітку $\mathbf{m_p}$.

Операції **БПВ** і **В** вибрані так, щоб алгоритм отримання ПОЛІЗ був якомога простішим.

Розглянемо реалізацію різних типів елементів списку циклу.

Елемент списку циклу типу арифметичного виразу

Оператор циклу має вигляд:

$$\mathbf{for\ d := a_1, a_2\ , a_3, ..., a_n\ do\ A;} \qquad (1.2)$$

де $\mathbf{a_1}$, $\mathbf{a_2}$, $\mathbf{a_3}$, ..., $\mathbf{a_n}$ – арифметичні вирази.

Даний цикл буде виконуватися за наступною схемою (Рисунок 1.25). Нехай, **i** – номер першої вільної робочої мітки в момент початку перекладу оператора циклу.

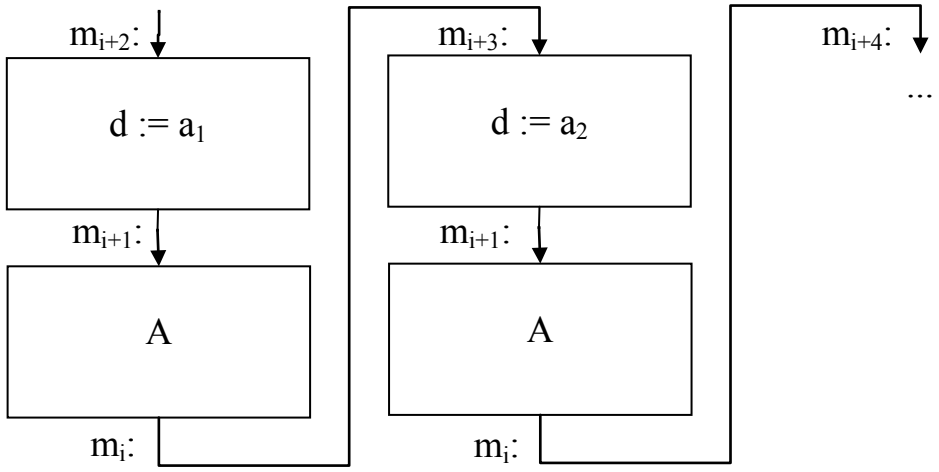


Рисунок 1.25 – Порядок виконання циклу з елементами списку типу арифметичного виразу

Після присвоєння **d := a_i** необхідно перейти на оператор **A**, а після нього повернутися в точку присвоєння наступного значення змінній циклу **d**. Для цього позначимо перелічені місця відповідними мітками. Позначимо міткою **m_i** точку, відповідну закінченню тіла циклу **A**, в цій точці буде генеруватися операція переходу для повернення за новим значенням **d**. Щоб зарезервувати місце для повернення, скористаємося операцією повернення **B**. Мітка **m_{i+1}** буде вказувати на початок тіла циклу. Для кожного **j**-го елемента списку циклу згенеруємо відповідну мітку **m_{i+j+1}**, що буде на нього вказувати. Отже мітки **m_{i+3}, m_{i+4}, … , m_{i+n+2}** – це адреси точок, в які треба повертатися після виконання тіла циклу **A** за значеннями **d,** що дорівнюють **a_2 , a_3, … , a_n**, відповідно. Початок оператора циклу помітимо **m_{i+2}**.

Оператор (1.2) дасть наступний польський запис (Рисунок 1.26), з урахуванням прийнятих раніше позначень: ПОЛІЗ(a_i) = **Пa_i**, ПОЛІЗ(A) = **ПA**.

Приклад 1.21

Розберемо для прикладу оператор **for d := a, b do A ;**

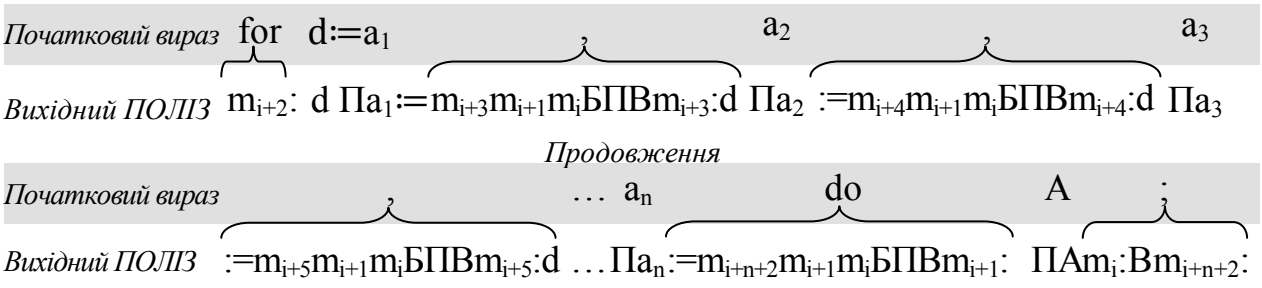


Рисунок 1.26 – Навантаження на службові слова для циклу зі списком елементів типу арифметичного виразу

В даному випадку (Приклад 1.21) список циклу містить два елементи **a** та **b**. Змінна циклу – **d**. Спочатку змінній циклу присвоюється значення **a**. Далі необхідно перейти на виконання тіла циклу, але при цьому слід запам'ятати поточне місце ПОЛІЗу, щоб повернутися туди за наступним значенням параметра циклу. Для цього використовуємо команду БПВ та помічаємо поточне місце ПОЛІЗу міткою $\mathbf{m_{i+3}}$. Команді **БПВ** передаємо наступні параметри (Рисунок 1.27):

1) перший – $\mathbf{m_{i+3}}$, що вказує на місце повернення, для присвоєння наступного значення параметра циклу (в даному випадку **Пb**);
2) другий – $\mathbf{m_{i+1}}$ – вказує на положення тіла циклу (оператора **ПА**);
3) третій – $\mathbf{m_i}$ – помічає місце операції повернення **В**, в якому при виконанні команди $\mathbf{m_{i+3}m_{i+1}m_iБПВ}$ буде генеруватися безумовний перехід на початок наступного елемента списку ($\mathbf{m_{i+3}БП}$). Це місце повинно знаходитися одразу за тілом циклу, тому після тіла циклу слід згенерувати в ПОЛІЗ $\mathbf{m_i:B}$.

Рисунок 1.27

Отриманий ПОЛІЗ забезпечить виконання тіла циклу **A** з параметром циклу **d=a** та повернення після цього на місце присвоєння параметру циклу значення **b**. Після присвоєння параметру циклу значення **b** слід знову виконати тіло циклу **A**, але оскільки наступного значення параметра циклу не передбачено,

після виконання **A** слід перейти на наступний за циклом оператор, це забезпечується використанням команди **БПВ** з тими самими параметрами, окрім першого. Тепер першим параметром **БПВ** є мітка $\mathbf{m_{i+4}}$, яка вказує на кінець оператора циклу.

Приклад 1.22

Побудуємо (Рисунок 1.28) ПОЛІЗ виразу **for i := a + b, 5 do k := i + 2;**. Позначаємо мітками $\mathbf{m_2}$ та $\mathbf{m_3}$ вирази визначення параметра циклу, міткою $\mathbf{m_1}$ – початок тіла циклу, міткою **m** – місце, в якому буде генеруватися повернення на чергове присвоєння параметра циклу, міткою $\mathbf{m_4}$ – вихід з циклу.

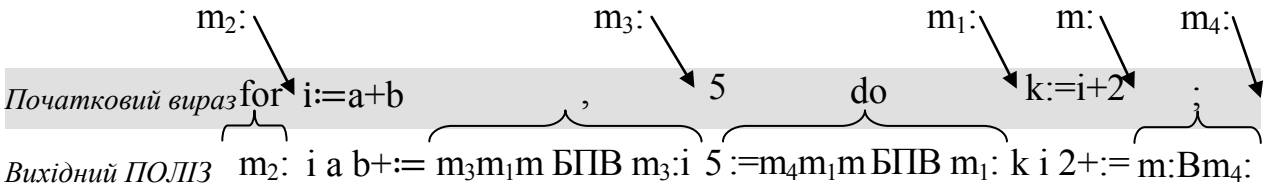


Рисунок 1.28

В даному прикладі першим значенням параметра циклу є результат арифметичного виразу **a+b**, другим – **5**, тілом циклу є операція **k ≔ i + 2** детально процес виконання можна зобразити у вигляді таблиці (Таблиця 1.29).

Таблиця 1.29. Приклад 1.22 виконання ПОЛІЗу

Резерв. ПОЛІЗ	m_2: i a b + ≔ m_3m_1m БПВ m_3: i 5 ≔ m_4m_1m БПВ m_1: k i 2 +≔m:B m_4:	
Крок 1	Команда m_2:	Дія -
	ПОЛІЗ	i a b + ≔ m_3m_1m БПВ m_3: i 5 ≔ m_4m_1m БПВ m_1: k i 2 +≔m:B m_4:
Крок 2	Команда i a b + ≔	Дія змінній **i** присвоєно значення виразу **a+b**
	ПОЛІЗ	m_3m_1m БПВ m_3: i 5 ≔ m_4m_1m БПВ m_1: k i 2 +≔m:B m_4:
Крок 3	Команда m_3m_1m БПВ	Дія замість **B** за адресою **m** вставляється $\mathbf{m_3БП}$, виконується перехід на $\mathbf{m_1}$
	ПОЛІЗ	m_3: i 5 ≔ m_4m_1m БПВ m_1: k i 2 +≔m: m_3БП m_4:

Продовження табл. 1.29

Крок 4	Команда m_1:	Дія -
	ПОЛІЗ	i 5 ≔ m_4m_1m БПВ m_1: k i 2 +≔m: m_3БП m_4:
Крок 5	Команда k i 2 +≔	Дія змінній **k** присвоєно значення виразу **2+i**, з урахуванням, що на даний момент **i=a+b**, тепер **k =a+b+2**
	ПОЛІЗ	m: m_3БП m_4:
Крок 6	Команда m:	Дія -
	ПОЛІЗ	m_3БП m_4:
Крок 7	Команда m_3БП	Дія Відтворення початкового ПОЛІЗу (рядок «Резерв. ПОЛІЗ») і перехід на мітку $\mathbf{m_3}$
	ПОЛІЗ	m_3: i 5 ≔ m_4m_1m БПВ m_1: k i 2 +≔m:B m_4:
Крок 8	Команда m_3:	Дія -
	ПОЛІЗ	i 5 ≔ m_4m_1m БПВ m_1: k i 2 +≔m:B m_4:
Крок 9	Команда i 5 ≔	Дія змінній **i** присвоєно **5**
	ПОЛІЗ	m_4m_1m БПВ m_1: k i 2 +≔m:B m_4:
Крок 10	Команда m_4m_1m БПВ	Дія замість **B** за адресою **m** вставляється $\mathbf{m_4БП}$, виконується перехід на $\mathbf{m_1}$
	ПОЛІЗ	m_1: k i 2 +≔m: m_4БП m_4:
Крок 11	Команда m_1:	Дія -
	ПОЛІЗ	k i 2 +≔m: m_4БП m_4:
Крок 12	Команда k i 2 +≔	Дія змінній **k** присвоєно значення виразу **2+i**, з урахуванням, що на даний момент **i=5**, тепер **k =7**
	ПОЛІЗ	m: m_4БП m_4:
Крок 13	Команда m:	Дія -
	ПОЛІЗ	m_4БП m_4:
Крок 14	Команда m_4БП	Дія Відтворення початкового ПОЛІЗу (рядок «Резерв. ПОЛІЗ») і перехід на мітку $\mathbf{m_4}$
	ПОЛІЗ	m_4:
Крок 15	Команда m_4:	Дія -
	ПОЛІЗ	

Елемент списку циклу типу перерахунку

Оператор такого циклу має наступний вигляд:

$$\textbf{for d := a}_1 \textbf{ while b}_1\textbf{, a}_2 \textbf{ while b}_2\textbf{, … , a}_n \textbf{ while b}_n \textbf{ do A;} \qquad (1.3)$$

де $\mathbf{a_1}$, $\mathbf{a_2}$ – арифметичні вирази;

$\mathbf{b_1}$, $\mathbf{b_2}$ – логічні вирази.

Як і раніше, позначимо мітками $\mathbf{m_i}$ та $\mathbf{m_{i+1}}$ точки кінця та початку тіла циклу **A**, а міткою $\mathbf{m_{i+2}}$ – початок оператора циклу. Позначимо мітками $\mathbf{m_{i+4}}$, $\mathbf{m_{i+6}}$, ... , $\mathbf{m_{i+2n}}$ точки початкового визначення параметра циклу ($\mathbf{d{:=}a_2}$, $\mathbf{d{:=}a_3}$), в які треба переходити після завершення елемента циклу. Мітки $\mathbf{m_{i+3}}$, $\mathbf{m_{i+5}}$, …, $\mathbf{m_{i+2n+1}}$ вказують на умови продовження циклу кожного елемента: $\mathbf{b_1}$, $\mathbf{b_2}$, …, $\mathbf{b_n}$ (Рисунок 1.29).

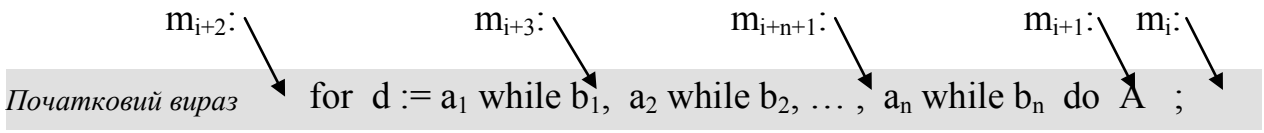


Рисунок 1.29

Блок-схема виконання такого циклу буде наступною (Рисунок 1.30).

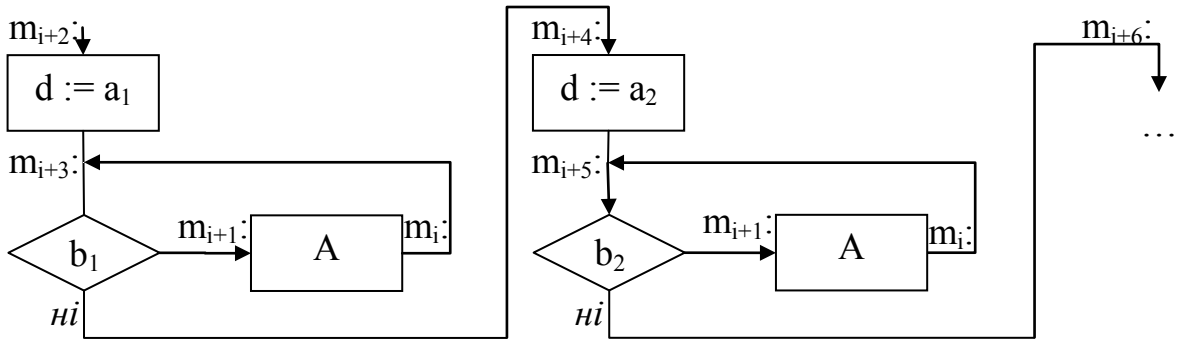


Рисунок 1.30 – Порядок виконання циклу з елементами списку типу перерахунку

Перевірка умов $\mathbf{b_1}$, $\mathbf{b_2}$, **...** потребує використання операцій **УПХ**. Знадобиться ще одна мітка $\mathbf{m_{i+2n+2}}$ для позначення остаточного виходу з циклу. Таким чином, оператор вигляду (1.3) транслюється в наступний ПОЛІЗ (Рисунок 1.31).

Початковий вираз for $d:=a_1$ while b_1 ;

Вихідний ПОЛІЗ m_{i+2}: $d\ Па_1:=$ m_{i+3}: $Пb_1$ m_{i+4}УПХ $m_{i+3}\ m_{i+1}\ m_i$ БПВ m_{i+4} : d

Продовження

Початковий вираз a_2 while b_2 ; …

Вихідний ПОЛІЗ $Па_2$ $:= m_{i+5}$: $Пb_2$ m_{i+6}УПХ $m_{i+5}m_{i+1}m_i$БПВ m_{i+6} : d …

Продовження

Початковий вираз a_n while b_n do A ;

Вихідний ПОЛІЗ $Па_n$ $:=m_{i+2n+1}$: $Пb_n$ m_{i+2n+2}УПХ$m_{i+2n+1}m_{i+1}m_i$БПВm_{i+1}: ПА m_i:Вm_{i+2n+2}:

Рисунок 1.31 – Навантаження на службові слова для циклу зі списком елементів типу арифметичного виразу

Приклад 1.23

Побудуємо ПОЛІЗ (Рисунок 1.32) виразу **for i:=3 while x<0, -1 while x>0 do x:=x+i;** .

Початковий вираз for $i := 3$ while $x < 0$;

Вихідний ПОЛІЗ m_2: $i\ 3 :=$ m_3: $x\ 0 <$ m_4 УПХ $m_3\ m_1$ m БПВ m_4 : i

Продовження

Початковий вираз -1 while $x > 0$ do $x := x+i$;

Вихідний ПОЛІЗ 1@ $:=m_5$: $x\ 0 >$ m_6 УПХ $m_5\ m_1$ m БПВ m_1 : $x\ x\ i+:=$ m:B m_6:

Рисунок 1.32

Якщо пронумерувати символи коду ПОЛІЗ (Таблиця 1.30), отримаємо наведені нижче значення міток (Таблиця 1.31), це дозволить видалити з коду фрагменти **мітка:** .

Таблиця 1.30. Пронумерований код ПОЛІЗу (Приклад 1.23)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
m_2:	i	3	:=	m_3:	x	0	<	m_4	УПХ	m_3	m_1	m	БПВ	m_4:	i	1	@	:=

Продовження таблиці 1.30

17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
m_5:	x	0	>	m_6	УПХ	m_5	m_1	m	БПВ	m_1:	x	x	i	+	:=	m:	B	m_6:

Таблиця 1.31. Значення міток (Приклад 1.23)

Мітка	Значення
m	31
m_1	26
m_2	1
m_3	4
m_4	13
m_5	17
m_6	32

ПОЛІЗ з урахуванням значень міток (Таблиця 1.31) набуде такого вигляду:

i 3 ≔ x 0 < m_4УПХ m_3m_1m БПВ i 1@≔ x 0 > m_6УПХ m_5m_1m БПВ x x 1 + ≔ B

Елемент списку циклу типу арифметичної прогресії

Оператор циклу зі списком елементів типу арифметичної прогресії має вигляд:

for d:=a_1 step h_1 to c_1, a_2 step h_2 to c_2, …, a_n step h_n to c_n do A;

де $\mathbf{a_k}$, $\mathbf{h_k}$, $\mathbf{c_k}$ ($\mathbf{k = 1..n}$) – арифметичні вирази.

Нехай $\mathbf{r_j}$ – чергова вільна робоча комірка, $\mathbf{m_i}$ – чергова вільна мітка. Позначимо мітками $\mathbf{m_i}$, $\mathbf{m_{i+1}}$ кінець і початок тіла циклу **A**. Як і раніше, на **for** вказує мітка $\mathbf{m_{i+2}}$. Для обробки першого елемента списку знадобиться дві мітки $\mathbf{m_{i+3}}$, $\mathbf{m_{i+4}}$, оскільки вони потрібні для циклу типу арифметичної прогресії (Рисунок 1.19). Позначимо міткою $\mathbf{m_{i+5}}$ точку присвоєння початкового значення параметра циклу для другого елемента циклу ($\mathbf{a_2}$). Також для другого елемента виділяються мітки $\mathbf{m_{i+6}}$, $\mathbf{m_{i+7}}$. Для інших елементів виділяємо мітки так само. Для останнього (n-ого) елемента списку буде виділено мітки $\mathbf{m_{i+3n-1}}$, $\mathbf{m_{i+3n}}$, $\mathbf{m_{i+3n+1}}$, відповідно (Рисунок 1.33). Тоді на вихід буде вказувати мітка $\mathbf{m_{i+3n+2}}$.

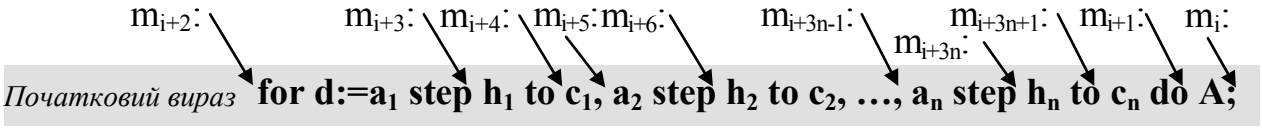


Рисунок 1.33 – Розташування міток для елементів арифметичної прогресії

Побудуємо блок-схему для початку циклу з елементами типу арифметичної прогресії (Рисунок 1.34).

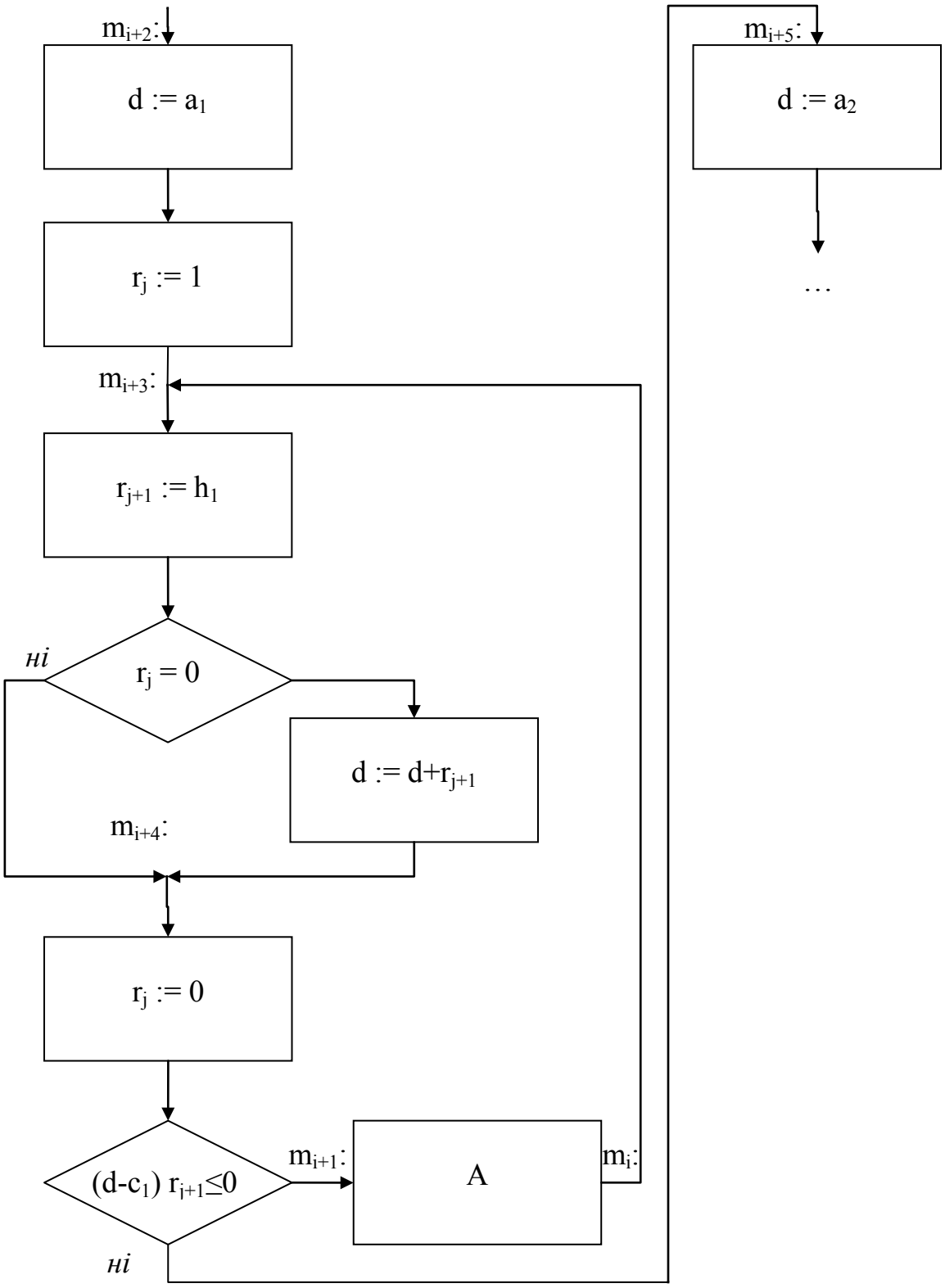


Рисунок 1.34 – Блок-схема циклу з елементами типу арифметичної прогресії

За блок-схемою сформуємо ПОЛІЗ (Рисунок 1.35).

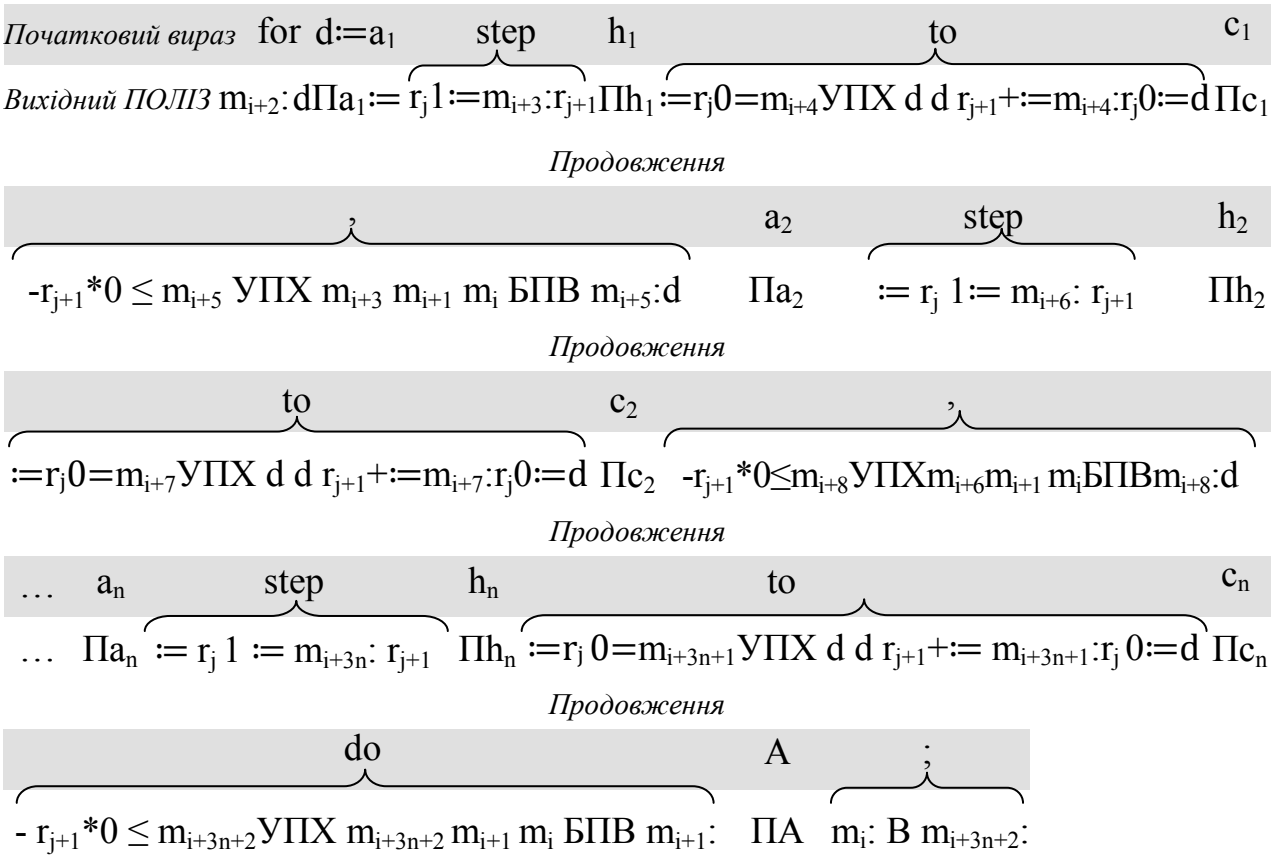


Рисунок 1.35 – Навантаження на службові слова для циклу зі списком елементів типу арифметичного виразу

Алгоритм побудови ПОЛІЗ оператора циклу зі списком елементів

При побудові ПОЛІЗу оператора циклу зі списком елементів особливим чином обробляються спеціальні символи: **for**, **:=**, **while**, **step**, **to**, **do**, **«,»** і кінець оператора **«;»** або **end**.

Під час трансляції будемо використовувати дві ознаки:

- ознаку циклу (ОЦ) з двома значеннями 0 і 1.
- ознаку типу елемента (ОТЕ) зі значеннями:
  - 0 – елемент списку циклу типу арифметичного виразу
  - 1 – елемент типу перерахунку.
  - 2 і 3 використовуються для елементів типу арифметичної прогресії.

Крім того, будемо використовувати одну стандартну змінну параметра циклу (ЗПЦ) для зберігання назви ідентифікатора параметра циклу. Розглянемо особливості обробки окремих символів.

1. Символ **for** грає роль відкриваючої дужки, тому має пріоритет 0. По символу **for** генеруються мітки $\mathbf{m_i}$, $\mathbf{m_{i+1}}$, $\mathbf{m_{i+2}}$, крім того
   a) у вихідний рядок записується: $\mathbf{m_{i+2}}$ **:**
   b) у вершину стека записується: **for** $\mathbf{m_{i+3}}$ $\mathbf{m_{i+1}}$ $\mathbf{m_i}$, де:
      - $\mathbf{m_i}$ – робоча мітка операції повернення **B**.
      - $\mathbf{m_{i+1}}$ – робоча мітка, що вказує на початок тіла циклу A.
      - $\mathbf{m_{i+3}}$ – робоча мітка ділянки програми, яка для елементів типу арифметичного виразу вказує на обчислення другого елемента списку циклу, типу перерахунку – на умову продовження циклу, типу арифметичної прогресії – на збереження в робочій комірці значення кроку;
   c) визначаються ознаки: **ОЦ=1**, **ОТЕ=0**, – тобто припускається, що перший елемент списку циклу має тип арифметичного виразу.

Примітка: пункт a виконується для загальності, існують деякі підвиди елементів списку, для яких потрібна дана мітка.

2. Символ **≔** має особливості, якщо зустрічається всередині оператора циклу. Якщо ОЦ=1, це означає, що останній запис ПОЛІЗу (ПОЛІЗ[k]) є ім'ям параметра циклу (для даного прикладу – **d**).
   a) символ **≔** у стек не заноситься;
   b) ідентифікатор параметра циклу (**d**) записується в стандартну змінну параметра циклу, тобто **ЗПЦ:=ПОЛІЗ[k]**;
   c) потім знімається ознака циклу, **ОЦ:=0**.
3. Символ **«,»** є закриваючою дужкою для попереднього виразу та має пріоритет 1, який забезпечує:
   a) виштовхування зі стека всіх знаків до запису **for** $\mathbf{m_p}$ $\mathbf{m_q}$ $\mathbf{m_r}$ виключно (цей запис було занесено в стек при обробці ключового слова **for**);
   b) подальші дії залежать від значення ознаки типу елемента ОТЕ (Таблиця 1.32). **Примітка:** у стовпчику «Запис у стек» вказано запис, який заноситься у вершину стека замість запису **for** $\mathbf{m_p}$ $\mathbf{m_q}$ $\mathbf{m_r}$;
   c) після цього ознаці типу елемента присвоюється нуль (**ОТЕ := 0**).

Таблиця 1.32

Значення ОТЕ	**Запис у вихідний ПОЛІЗ**	**Запис у стек**
0 – арифметичний вираз	:= m_p m_q m_r БПВ m_p: d	for m_{p+1} m_q m_r
1 – перерахунок	m_{p+1} УПХ m_p m_q m_r БПВ m_p: d	for m_{p+2} m_q m_r
2 – помилка: після step немає to	*Занесення в таблицю помилок*	
3 – арифметична прогресія	– r_{j+1} * 0 ≤ m_{p+2} УПХ m_p m_q m_r БПВ m_{p+2}: d	for m_{p+3} m_q m_r зменшити лічильник робочих комірок на два

4. Слово **do** є закриваючою дужкою для попереднього виразу та всього заголовка, отже, воно має пріоритет 1. За **do** слід виконати наступні дії:
   a) усі знаки до виразу **for m_p m_q m_r** (виключно) виштовхуються зі стека на вихід;
   b) подальші дії залежать від типу елемента циклу (Таблиця 1.33). **Примітка:** у стовпчику «Запис у стек» вказано запис, який заноситься у вершину стека замість запису **for m_p m_q m_r**;
   c) **ОТЕ := 0**.

Таблиця 1.33

Значення ОТЕ	**Запис у вихідний ПОЛІЗ**	**Запис у стек**
0 – арифметичний вираз	:= m_p m_q m_r БПВ m_p:	for m_{p+1} m_q m_r
1 – перерахунок	m_{p+1} УПХ m_p m_q m_r БПВ m_p:	for m_{p+1} m_q m_r
2 – помилка: після step немає to	*Занесення в таблицю помилок*	
3 – арифметична прогресія	– r_{j+1} * 0 ≤ m_{p+2} УПХ m_p m_q m_r БПВ m_q:	for m_{p+2} m_q m_r зменшити лічильник робочих комірок на два

5. Символ **while** є закриваючою дужкою для попереднього виразу в елементі списку циклу типу перерахунку і має пріоритет 1, відповідно до нього забезпечуються наступні дії:

   a) зі стека виштовхуються всі записи до **for** $\mathbf{m_p}$ $\mathbf{m_q}$ $\mathbf{m_r}$ виключно;

   b) потім у вихідний рядок заноситься запис **:=** ;

   c) ознака типу елемента приймає значення 1 (**OTE := 1**).

Примітка: Перед цим ознака типу елемента повинна була мати значення 0, інше значення свідчить про помилку.

6. Символ **step** є закриваючою дужкою для попереднього виразу в елементі списку циклу типу арифметичної прогресії, має пріоритет 1, генерує дві робочі комірки та забезпечує виконання таких дій:

   a) зі стека виштовхуються всі записи до **for** $\mathbf{m_p}$ $\mathbf{m_q}$ $\mathbf{m_r}$;

   b) у вихідний рядок заноситься запис **:=** $\mathbf{r_j}$ **1 :=** $\mathbf{m_p}$ **:** $\mathbf{r_{j+1}}$

   c) ознака типу елемента приймає значення 2 (**OTE := 2**).

Примітка: перед цим ознака типу елемента повинна була мати значення 0, інше значення свідчить про помилку: неправильно записаний попередній елемент списку або неправильно записаний заголовок, наприклад відсутнє слово **for**.

7. Символ **to**, як і **step**, є закриваючою дужкою і має пріоритет 1, відповідно до **to** виконуються дії:

   a) зі стека виштовхуються всі записи до **for** $\mathbf{m_p}$ $\mathbf{m_q}$ $\mathbf{m_r}$ (виключно);

   b) у вихідний рядок заноситься запис **:=** $\mathbf{r_j}$ **0 =** $\mathbf{m_{p+1}}$ **УПХ d d** $\mathbf{r_{j+1}}$ **+ :=** $\mathbf{m_{p+1}}$ **:** $\mathbf{r_j}$ **0 := d**;

   c) ознака типу елемента приймає значення 3 (**OTE := 3**).

Примітка: перед цим ознака типу елемента повинна була мати значення 2, інше значення свідчить про помилку: перед **to** немає **step**.

8. **Кінець оператора** (символи **;** або **end**). Символи кінця оператора спочатку обробляються згідно зі своїми пріоритетами, далі:

   a) зі стека виштовхуються всі записи до **for** $\mathbf{m_p}$ $\mathbf{m_q}$ $\mathbf{m_r}$. Цей запис в стеку позначає кінець даного оператора циклу;

b) у вихідний рядок заноситься запис $\mathbf{m_r}$**: В** $\mathbf{m_p}$**:**

c) після цього запис **for** $\mathbf{m_p}$ $\mathbf{m_q}$ $\mathbf{m_r}$ видаляється зі стека, й виштовхування записів зі стека продовжується.

Примітка: запис виду **for** $\mathbf{m_p}$ $\mathbf{m_q}$ $\mathbf{m_r}$ може зустрічатися кілька разів поспіль (при обробці вкладених циклів), тоді кожного разу у вихідний рядок заноситься запис виду $\mathbf{m_r}$**: В** $\mathbf{m_p}$**:**.

Приклад 1.24

Нехай, необхідно перевести в ПОЛІЗ оператор

for x := y, 1 step 2 to 8 do y ≔ y +x.

Визначимо положення ключових міток (Рисунок 1.36).

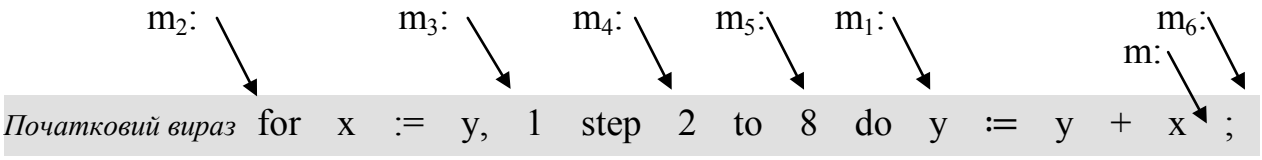


Рисунок 1.36

Виконаємо переведення в ПОЛІЗ відповідно до алгоритму, представленого вище (Таблиця 1.34).

Наведемо остаточний ПОЛІЗ (Рисунок 1.37).

Початковий вираз	for	x ≔ y	,	1	step	
Вихідний ПОЛІЗ	m_{i+2}:	x y ≔	:= m_3 m_1 m БПВ m_p: x	1	:= r_1 1:= m_4: r_2	

Продовження

Початковий вираз	2	to	8
Вихідний ПОЛІЗ	2	:= r_1 0 = m_5 УПХ x x r_2 + := m_5: r_1 0 := x	8

Продовження

Початковий вираз	do	y:=y+x	;
Вихідний ПОЛІЗ	- r_2*0 ≤ m_6 УПХ m_4 m_1 m БПВ m_1:	y y x+:=	m B m_6:

Рисунок 1.37 – ПОЛІЗ циклу for x := y, 1 step 2 to 8 do y ≔ y +x

Таблиця 1.34. Переклад в ПОЛІЗ вкладеного циклу

ПОЛІЗ	m_2 :	x		y	:= m_3 m_1 m БПВ m_3 : x	1	:= r_1 1 := m_4 : r_2	2	:= r_1 0 = m_5 УПХ x x r_2 + := m_5: r_1 0 := x	8	– r_2 * 0 ≤ m_6 УПХ m_4 m_1 m БПВ m_1 :	y		y		x	+ := m : B m_6 :
ЗПЦ	-	-	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
ОЦ	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ОТЕ	0	0	0	0	0	0	2	2	3	3	0	0	0	0	0	0	0
СТЕК															+	+	
													:=	:=	:=	:=	
	for m_3 m_1 m	for m_3 m_1 m	for m_3 m_1 m	for m_3 m_1 m	for m_4 m_1 m	for m_4 m_1 m	for m_4 m_1 m	for m_4 m_1 m	for m_4 m_1 m	for m_4 m_1 m	for m_6 m_1 m	for m_6 m_1 m	for m_6 m_1 m	for m_6 m_1 m	for m_6 m_1 m	for m_6 m_1 m	
Вхід	for	x	:=	y	,	1	step	2	to	8	do	y	:=	y	+	x	;

де ЗПЦ – змінна параметра циклу, в даному прикладі – **x**;

ОЦ – ознака циклу, яка використовується для передачі імені параметра циклу з коду ПОЛІЗ в комірку параметра циклу;

ОТЕ – ознака типу елемента списку циклу (в початковому стані має значення 0, яке надалі змінюється в залежності від розпізнаного типу елемента списку циклу);

СТЕК поділено на комірки, перший запис **for m_3 m_1 m** модифікується по мірі обробки вхідного ланцюжка та використовується для генерації відповідних фрагментів ПОЛІЗ.

Вхід – вхідний ланцюжок, що розбирається.

1.2. Тріади та тетради як проміжні форми подання програми

Окрім польського інверсного запису існують також інші форми внутрішнього подання програми, що транслюється. До них відносяться тріади та тетради [15; 16].

Тетрада – це кортеж, що складається з чотирьох елементів:

<оператор>,<операнд1>,<операнд2>,<результат>

Таким чином, арифметичний вираз з однією операцією множення **a*b** у формі тетради буде мати наступний вигляд:

*, a, b, t

Якщо вираз містить кілька арифметичних операцій, він буде описуватися кількома послідовними тетрадами, наприклад, вираз **(a*b)+(c*d)** перетвориться на:

*, a, b, t1
*, c, d, t2
+, t1, t2, t3

Тетради розташовуються в порядку виконання. Їхнім недоліком є велика кількість проміжних змінних, для нескладного виразу з попереднього прикладу використано дві проміжні змінні: **t1**, **t2**. Такого недоліку не мають тріади.

Тріади – є кортежами з трьох елементів, що мають наступну форму:

<оператор>, <операнд1>,<операнд2>

Тріади відрізняються від тетрад тим, що не мають поля для результату. Для того, щоб використати результат обчислення тріади в наступних обчислен-

нях використовують посилання на відповідну тріаду. Наприклад, вираз **a+b*c** у формі тріад запишеться як:

(1) *, b, c
(2) +, a, (1)

де **(1)** – це посилання на результат 1-ої тріади.

Вираз **1+b*c** запишеться як:

(1) *, b, c
(2) + 1, (1)

Розглянемо детальніше тетради. Принцип тетрад може бути поширений на будь-які оператори. Наприклад, можна встановити таку (Таблиця 1.35) відповідність записів операцій у вигляді ПОЛІЗу й у формі тетрад.

Таблиця 1.35. Основні операції в формі тетрад

ПОЛІЗ	**Тетради**
a b +	+, a, b, t_j
@ a	-, a, , t_j
a b :=	:=, b, , a
m_1 БП	БП, m_1 ,
a m_1 УПХ	УПХ, m_1, a

Тетради можуть бути згенеровані в ході будь-якого алгоритму синтаксичного розбору.

1.2.1. Генерація тетрад при висхідному синтаксичному розборі

Розглянемо процес отримання тетрад при застосуванні висхідного розбору.

Для того, щоб побудувати тетради при висхідному розборі, необхідно задати семантичні процедури, відповідні правилам граматики (Таблиця 1.36).

Таблиця 1.36. Семантичні процедури граматики арифметичного виразу для побудови тетрад

№	Правило граматики	Семантична процедура
1	E1::=E	E1.SEM := E.SEM
2	E::=E+T1	j := j+1 ENTER(“+”, E.SEM, T1.SEM, t_j) E.SEM := t_j
3	E::=E-T1	j:=j+1 ENTER(“-”, E.SEM, T1.SEM, t_j) E.SEM:=t_j
4	E::=T1	E.SEM:=T1.SEM
5	E::= -T1	j:=j+1 ENTER(“-”, T1.SEM, , t_j) E.SEM:=t_j
6	T1::=T	T1.SEM:=T.SEM
7	T::=T*F	j:=j+1 ENTER(“+”, T.SEM, F.SEM, t_j) T.SEM:=t_j
8	T::=T/F	j:=j+1 ENTER(“/”, T.SEM, F.SEM, t_j) T.SEM:=t_j
9	T::=F	T.SEM:=F.SEM
10	F::=(E1)	F.SEM:=E1.SEM
11	F::=i	F.SEM:=i.SEM

де **j** – номер додаткового імені для зберігання результату операції;

ENTER – процедура, що генерує тетраду та відповідне ім'я результату.

В процесі розбору будемо пов'язувати імена ідентифікаторів (та, відповідно, їхні значення) з нетерміналами, на які їх було замінено під час згортки.

Приклад 1.25

Отримаємо тетради відповідно до вказаних семантичних процедур, наприклад, для арифметичного виразу: **a*(-b+c)**.

Для цього виразу мають бути згенеровані наступні тетради:

-,	b,	,	t1
+,	t1,	c,	t2
*,	a,	t2,	t3

Побудуємо дерево розбору приведеного виразу (Рисунок 1.38) відповідно до граматики (Таблиця 1.36).

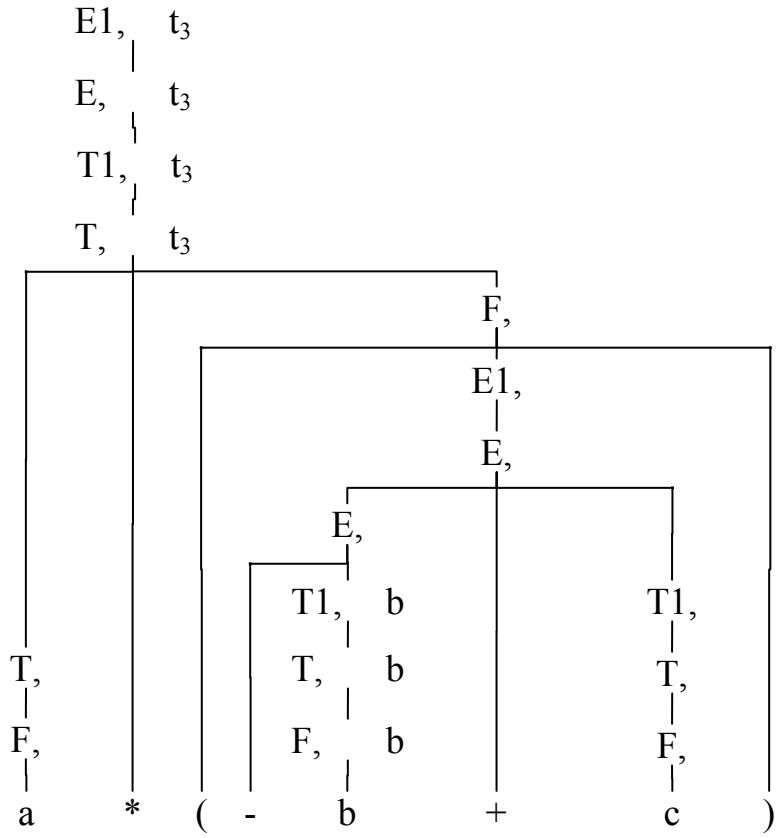


Рисунок 1.38 – Дерево розбору

При побудові дерева розбору нетермінальним символам ставиться у відповідність семантичний еквівалент, тобто імена ідентифікаторів вхідної програми або додаткових змінних для зберігання проміжних результатів. При заміні одного нетермінала іншим (за правилами № 1, 4, 6, 9) семантичний

еквівалент передається новому нетерміналу без зміни. Якщо в основу входить операція, то згортці (тобто заміні основи новим нетерміналом) ставиться у відповідність виконання цієї операції, а новому нетерміналу ставиться у відповідність новий семантичний еквівалент, що є результатом даної операції. Ідея призначення семантичного еквівалента реалізована у вигляді семантичних підпрограм, які відповідають правилам граматики (Таблиця 1.36).

Опишемо покрокову реалізацію висхідного розбору з паралельною побудовою тетрад за допомогою семантичних процедур у вигляді таблиці (Таблиця 1.37). На кожному кроці будемо записувати вміст стека (основу будемо виділяти жирним шрифтом, а нетермінали з відповідними семантичними еквівалентами, що складають неподільні елементи стеку, будемо виділяти сірим фоном), відношення між останнім елементом стека та першим елементом вхідного ланцюжка (стовпчик «Відн.»), вхідний ланцюжок («Вхід»), застосовані семантичні процедури та тетради («Сем.п/п і тетради»).

Таблиця 1.37. Побудова тетрад для виразу a*(-b+c)

Стек	**Відн.**	**Вхід**	**Сем. п/п і тетради**
#	·>	a*(-b+c)#	j = 0
# **a**	·>	*(-b+c) #	
# **F,a**	·>	*(-b+c) #	F.sem := a
# T,a	≐	*(-b+c) #	T.sem := F.sem = a
# T,a *	<·	(-b+c) #	
# T,a * (	<·	-b+c) #	
# T,a * (-	<·	b+c) #	
# T,a * (– **b**	·>	+c) #	
# T,a * (– **F,b**	·>	+c) #	F.sem := b
# T,a * (– **T,b**	·>	+c) #	T.sem := F.sem = b
# T,a * (– **T1,b**	·>	+c) #	T1.sem := T.sem = b

Продовження табл. 1.37

Стек	Відн.	Вхід	Сем. п/п і тетради
# T,a * (E, t1	<·	+c) #	j = 0 + 1 = 1 -, b, , t1 E.sem := t1
# T,a * (E, t1 +	<·	c) #	
# T,a * (E, t1 + **c**	·>	) #	
# T,a * (E, t1 + **F,c**	·>	) #	F.sem := c
# T,a * (E, t1 + **T,c**	·>	) #	T.sem := F.sem = c
# T,a * (**E, t1 + T1,c**	·>	) #	T1.sem := T.sem = c
# T,a * (**E, t2**	·>	) #	j = 1 + 1 = 2 +, t1, c, t2 E.sem := t2
# T,a * (E1, t2	≐	) #	E1.sem := E.sem = t2
# T,a * **(E1, t2)**	·>	#	
# **T,a * F, t2**	·>	#	F.sem := E1.sem = t2
# **T,t3**	·>	#	j = 2 + 1 = 3 *, a, t2, t3 T.sem := t3
# **T1,t3**	·>	#	T1.sem := T.sem = t3
# **E,t3**	·>	#	E.sem := T1.sem = t3
# E1,t3	·>	#	E1.sem := E.sem = t3

Початкове значення **j:=0**. В процесі розбору отримаємо:

для j:=1	-,	b,	,	t1	E.SEM:=t1
для j:=2	+,	t1,	c,	t2	E.SEM:=t2
для j:=3	*,	a,	t2,	t3	T.SEM:=t3

1.2.2. Генерація тетрад при рекурсивному спуску

Тетради, як і інші проміжні представлення програм, можна також отримати в результаті синтаксичного аналізу методом рекурсивного спуску. Розглянемо, як це виконується, на прикладі розбору арифметичного виразу.

Приклад 1.26

Граматика арифметичного виразу для реалізації рекурсивного спуску:

1. Z ::= E
2. E ::= (- | ^) T { (+ | –) T }
3. T ::= F { (* | /) F }
4. F ::= i | (E)

Паралельно з виконанням процедур синтаксичного розбору будемо виконувати процедури передачі семантичних еквівалентів кожному нетерміналу та генерувати відповідні тетради. Семантичний еквівалент будемо подавати як параметр відповідної процедури розбору, що має спеціальний тип Semantica.

Приведемо схеми всіх чотирьох процедур рекурсивного спуску відповідно до правил граматики арифметичного виразу (Рисунок 1.39-1.42).

Нехай, i – номер поточної робочої змінної для збереження результатів тетрад (початкове значення i = 0), t – масив робочих змінних результатів тетрад. Вважаємо, що вже існує процедура AddTetrad(), яка додає у вихідний список тетрад нову тетраду, сформовану з чотирьох вхідних параметрів.

Для двомісних операцій вводимо змінні типу Semantica для збереження семантичних еквівалентів кожного з операндів (в функції E це змінні Tsem1 та Tsem2, а в функції T – Fsem1, Fsem2). Після додавання тетради, що відповідає двомісній операції, результат цієї тетради зберігаємо в змінній, яка відповідає першому операнду (Tsem1 або Fsem1), оскільки другий операнд є необов'язковим. Тоді в кінці підпрограми остаточний семантичний еквівалент відповідного нетермінала буде зберігатися в змінній для першого операнда.

```
function Z(var Zsem : Semantica): Boolean;
var Esem : Semantica;
begin
  Перейти до наступної лексеми
  result := E(Esem)~1/1; //виклик процедури, що відповідає
  Zsem := Esem;     //другому правилу граматики}
end; // of Z
```

Рисунок 1.39 – Схема підпрограми для нетермінала **Z**

```
function E(var Esem : Semantica): Boolean;
var Tsem1, Tsem2 : Semantica;
begin
//Обчислення семантичного еквівалента першого
//операнда двомісної арифметичної операції ±
  if lex = "-" then
  begin
    Перейти до наступної лексеми
    Result := T(Tsem1)~6/6;
    i := i + 1;
    AddTetrad("-", Tsem1, NULL, t[i]);
    Tsem1 := t[i];
  end else Result := T(Tsem1)~2/2;
  while lex = "+" or lex = "-" do
  begin
//Обчислення семантичного еквівалента наступних операндів
    Перейти до наступної лексеми
    Result := T(Tsem2)~8/8;
    i := i + 1;
    AddTetrad("+"|"-", Tsem1, Tsem2, t[i]);
    Tsem1 := t[i];
  end;
  Esem := Tsem1;
end // of E
```

Рисунок 1.40 – Схема підпрограми для нетермінала **E**

```
function T(var Tsem : Semantica) : Boolean;
var Fsem1, Fsem2 : Semantica;
begin
  Result := F(Fsem1)~3/3~7/7~9/9;
  Перейти до наступної лексеми
  while lex = "*" or lex = "/" do
  begin
    Перейти до наступної лексеми
    Result := F(Fsem2)~4/4;
    i := i + 1;
    AddTetrad("*"|"/", Fsem1, Fsem2, t[i]);
    Fsem1 := t[i];
  end;
  Tsem := Fsem1;
end // of T
```

Рисунок 1.41 – Схема підпрограми нетермінала **T**

```
function F (var Fsem : Semantica) : Boolean;
var Esem : Semantica;
begin
  Result := true;
  if lex = i then Fsem := i.sem
  else if lex = "(" then
  begin
    Перейти до наступної лексеми
    Result := E (Esem)~5/5;
    if lex = ")" then Fsem := Esem else Result := false;
  end else Result := false;
end //of F
```

Рисунок 1.42 – Схема підпрограми нетермінала **F**

Приклад 1.27

Розглянемо як приклад генерацію тетрад при розборі виразу:

a * (– b + c).

Побудуємо схему викликів підпрогам (Рисунок 1.39-1.42 номери викликів позначені як верхні індекси з тільдою «~», повернення зі слешем «/») і повернення семантичного еквівалента (Рисунок 1.43).

На схемі (Рисунок 1.43) словом «call» позначено виклики підпрограм рекурсивного спуску. Для зручності виклики нумеруються. На сірому фоні зображені дії по присвоєнню семантичних еквівалентів. Надходження нової лексеми відображено у вигляді квадратів, всередині яких знаходяться зчитані лексеми. Тетради, що генеруються, графічно представлені у вигляді прямокутників з округленими кутами. Значення семантичного еквівалента, що є вихідним параметром підпрограм рекурсивного спуску, при поверненні вказується нахиленим шрифтом. Прямі стрілки, що йдуть зверху-вниз, відображають виклик підпрограм та інших дій. Круглими стрілками зображено повернення. Пунктирні стрілки (зліва-направо) вказують на послідовні дії, наприклад: після повернення до сьомого виклику зчитується лексема.

Таким чином, схема відображає наступну послідовність дій. Керуюча програма викликає підпрограму Z (0-виклик). Підпрограма Z викликає функцію

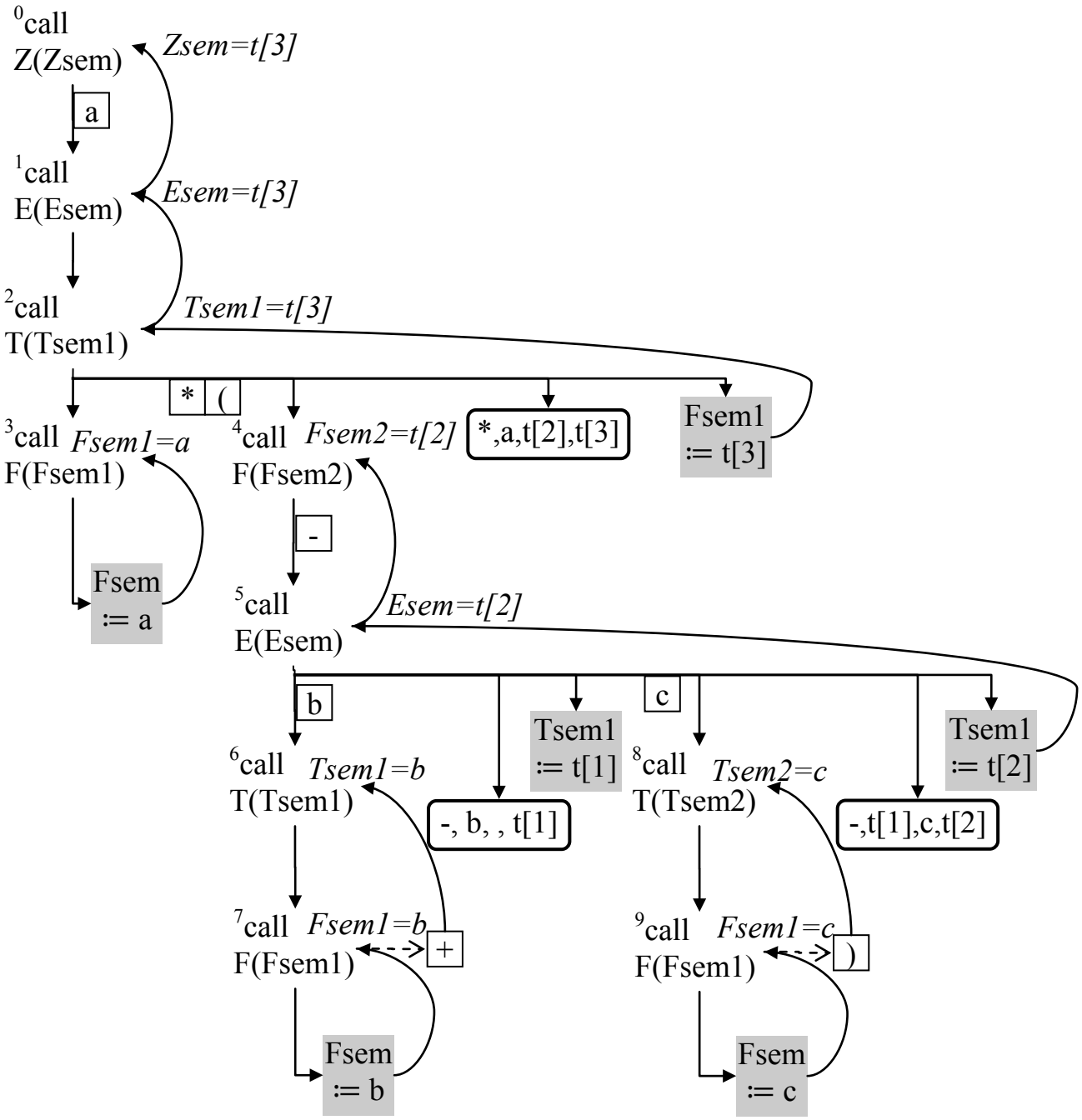


Рисунок 1.43 – Схема виклику підпрограм рекурсивного спуску при розборі виразу a*(-b+c)

E (виклик 1) з вихідним параметром Esem. Функція E зчитує наступну лексему **a** та викликає функцію T (виклик 2) з вихідним параметром Tsem1. Підпрограма T, в свою чергу, викликає функцію F (виклик 3), очікуючи від неї значення параметра Fsem1. Всередині функції F обробляється вхідна лексема **a** і внутрішньому параметру Fsem присвоюється семантичний еквівалент **a**. Після завершення обробки третього виклику (підпрограми F) виконується повернення

в функцію T, яка здійснила третій виклик. При поверненні значення внутрішнього параметра (Fsem) передається зовнішньому параметру (Fsem1) функції F, внаслідок чого змінна Fsem1 приймає значення **a**. Після повернення всередині функції T зчитується наступна лексема, оскільки це «*», розпочинається цикл, в якому зчитується ще одна лексема «**(**» і знов запускається функція F (виклик 4). Оскільки вхідна лексема – «**(**», F починає обробляти дужкову конструкцію, при цьому зчитується наступна лексема «-» і викликається підпрограма E, яка знову викликає T та F, в процесі зчитуються лексеми **b** та «+». Коли управління повертається до E, вона виділяє першу тетраду: «-, b, , t[1]». У змінну Tsem1, що зберігає семантичний еквівалент першого доданка арифметичного виразу, заноситься значення результату виділеної тетради **t[1]**.

Оскільки поточна лексема – це «+», на виконання запускається цикл. В середині циклу зчитується наступна лексема **c**, і в підпрограмі E другий раз викликається функція T (виклик 8) для обробки другого доданку. Функція T викликає F (виклик 9), де відповідному семантичному еквіваленту присвоюється значення **c**. По завершенні функції F з вхідного потоку береться наступна лексема «**)**», після чого потік управління передається назад функції E, і семантичний еквівалент другого доданку приймає значення **c**. Далі функція E виділяє тетраду для операції додавання: «-,t[1],c,t[2]», – і в змінну, де зберігається семантичний еквівалент першого доданку заноситься результат цієї тетради **t[2]** для того, щоб в кінці даної підпрограми присвоїти це значення остаточному семантичному еквіваленту. Оскільки поточна лексема не є «+» або «**-**», цикл завершується (було виконано лише один прохід циклу) і остаточний семантичний еквівалент **t[2]** повертається (повернення виклику 5) в змінну семантичного еквіваленту виразу, що в дужках. Оскільки поточна лексема є закриваючою дужкою «**)**» підпрограма завершується успіхом, управління повертається (повернення виклику 4) до функції T, а семантичний еквівалент другого множника Fsem2 приймає значення **t[2]**. У підпрограмі T виділяється тетрада, що відповідає операції множення: «*,a,t[2],t[3]». Оскільки поточний символ не є «*» або «/», цикл підпрограми T завершується. Результат цієї тетради **t[3]** стає остаточ-

ним семантичним еквівалентом функції T, який передається спочатку у відповідь на виклик 1 підпрограмі E (цикл ітеративного виразу «{ (+ | –) T }» в цій підпрограмі не починається, через те що вхідна лексема «)», а не «+» чи «-»), потім Z як відповідь на виклик 0.

Існують також інші форми внутрішнього подання програм, наприклад, синтаксичні дерева [17] різноманітні лінеаризовані форми подання [16]. У випадку необхідності можуть бути створені додаткові спеціалізовані форми подання програм [18]. Усі форми внутрішнього подання програми повинні легко генеруватися, їх переклад на цільову мову або безпосереднє виконання інтерпретатором також має бути простим, вони повинні бути зручними для виконання машинонезалежної оптимізації [19; 20; 21].

1.3. Завдання для самоконтролю

Завдання 1.1. Визначити типи команд, що використовуються для виконання програм.

Завдання 1.2. Дайте формальне визначення постфіксного запису.

Завдання 1.3. Визначити правила ПОЛІЗ.

Завдання 1.4. Переведіть в ПОЛІЗ наступні арифметичні вирази:

а) $-a+2*b$;

б) $-(a+2*b)$;

в) $-(a+2)*b$;

г) $a+2*b\text{^}2-5++b$;

ґ) $a+(2*b)\text{^}2-5+b$;

д) $a+(2*b)\text{^}2-(5+b)$;

е) $b\text{^}2-4*a*c\text{^}1/2$;

є) $(b\text{^}2-4*a*c)\text{^}1/2$;

ж) $(b\text{^}2-4*a*c)\text{^}(1/2)$;

з) $((b\text{^}2)-4*(a*c))\text{^}(1/2)$;

и) $\dfrac{5+12*b}{5-12*b}$.

Завдання 1.5. Сформувати граматику простого передування та визначити необхідні семантичні підпрограми для мови арифметичного виразу, що містить:

а) бінарні та унарні операції +, -;

б) бінарні та унарні операції +, -, бінарні операції *, /;

в) бінарні операції +, -, *, /, ^;

г) бінарні та унарні операції +, -, бінарні операції *, /, ^;

ґ) бінарні та унарні операції +, - та дужки: (,);

д) бінарні та унарні операції +, -, бінарні операції *, / та дужки: (,);

е) бінарні операції +, -, *, /, ^ та дужки: (,);

є) бінарні та унарні операції +, -, бінарні операції *, /, ^ та дужки: (,);

Завдання 1.6. Встановіть, які граматики з завдання 1.5 припускають вирази з завдання 1.4.

Завдання 1.7. Побудувати ПОЛІЗ для запису:

а) присвоєння: i := b+2^c;

б) присвоєння, що має результат: i := j := b+2^c;

в) операцій інкременту та декременту: i++, ++i, i--, --i;

г) операцій: i += 2, i *= 2;

ґ) операцій виводу та вводу: ≪ “i = ” ≪ i , ≫ a ≫ b ≫ c;

д) операцій виводу та вводу: write(a, b, c), read(b, c);

Завдання 1.8. Сформувати граматику простого передування та семантичні підпрограми для мови:

а) логічних виразів, що містить логічні операції AND, OR, NOT, а їхніми операндами можуть бути відношення <, >, = між двома ідентифікаторами або константами;

б) логічних виразів, що містить логічні операції AND, OR, NOT, а їхніми операндами можуть бути відношення <, >, = між двома арифметичними виразами з завдання 1.5;

в) логічних виразів, що містить (С-подібні) логічні операції &, &&, |, ||, !, а їхніми операндами можуть бути відношення <, >, = між двома ідентифікаторами або константами;

г) оператора присвоєння ідентифікатору деякого арифметичного виразу (з завдання 1.5);

ґ) оператора присвоєння ідентифікатору деякого арифметичного виразу (з завдання 1.5), що повертає результат;

д) що містить оператори префіксного та постфіксного інкременту та декременту;

е) що містить оператори складеного присвоєння: +=, -=, *=, /=;

є) оператора присвоєння ідентифікатору результату відношення двох арифметичних виразів (з завдання 1.5);

ж) оператора присвоєння ідентифікатору результату логічного виразу (з завдання 1.8 а, б);

з) операторів вводу/виводу: write(<список ідентифікаторів>), read(<список ідентифікаторів>), що можуть містити порожній список ідентифікаторів;

и) операторів почергового вводу/виводу з завдання 1.7 ґ.

Завдання 1.9. Визначити пріоритети операцій та особливості обробки усіх операторів та службових слів для мов з Завдання 1.8, за умови використання алгоритму Дейкстри для побудови ПОЛІЗ.

Завдання 1.10. Виконати побудову ПОЛІЗ для записів, що відповідають мовам Завдання 1.8, використовуючи результати Завдання 1.9:

а) (s < 0 AND NOT x = 0) OR s > 0;

б) логічні вирази:

б.а) (a < a + b AND a > a – b) OR (a > a + b AND a < a – b);

б.б) (s > 0 AND NOT s – s / +1 = 0 AND x > 0) OR -s < 0;

б.в) NOT ((a > b^2*7 AND a < b) OR a > b);

б.г) + a > b AND NOT (a > b^2*7 AND a < b);

б.ґ) (a+(b – c) < a + b AND a > a – b OR -c – a > a + b;

б.д) NOT ((a > b*(2+c) AND -a < b) OR +a > b);

б.е) (NOT (a > b^(c*2+1) AND a < b)) OR a > b;

б.є) (a + b^-(c*2+1) AND a < b) OR NOT a = b;

в) s > a & s < 0 || !s=0;

г) оператори присвоєння:

г.а) a ≔ a + b – 10 +c;

г.б) a ≔ a – b / c * 100;

г.в) a ≔ b^2*7+5;

г.г) a ≔ -b + b*2^3*a;

г.ґ) a ≔ a + (1+b – 10) +c;

г.д) a ≔ -b * (-(2+c)) + 3*a*c;

г.е) a ≔ b^(c*2+1)*c;

г.є) a ≔ -a + b^-(c+2*a);

ґ) оператори присвоєння з результуючим значенням:

ґ.а) a ≔ b ≔ a + b;

ґ.б) a ≔ b ≔ b / c * 100 + 10;

ґ.в) b ≔ a ≔ a*b^7+5;

ґ.г) b≔ a ≔ -b – b^3*a + 1;

ґ.ґ) a ≔ b ≔ c ≔ a + (10 – b) +15;

ґ.д) x ≔ a ≔ -(b * (8 – a) + 3)*a*c;

ґ.е) b ≔ a ≔ 5*(1+b)^(c*2+1) / (12 * c – 5);

ґ.є) a ≔ a ≔ -a^(-c+2*a) / (5 * c);

д) i++; --i; j ≔ ++i; j ≔ x * (++i);

е) a += b / c * 100; p *= p;

є) оператори присвоєння відношень:

є.а) a ≔ a + b > a – c;

є.б) s ≔ s – s / 15 < a + 20;

є.в) a ≔ b^2*7 < a + b*5;

є.г) b ≔ – a > b + b^a*7;

є.ґ) b ≔ a+(b – c) < a + b;

є.д) c ≔ b*2+c = -a*(2+c);

є.е) a ≔ b^(c*2+1) < b;

є.є) a ≔ b^-(c*2+1) > (b+12)^2;

ж) оператори присвоєння логічних виразів:

ж.а) a ≔ (a > b OR (a < c AND c > 10) AND b < 0) OR (b > 0 AND (NOT b = c OR c = 10));

ж.б) оператора присвоєння ідентифікатору результату логічного виразу (з завдання 1.10 б);

з) read(f, b); write(); read(); write(a, n, b);

и) операторів почергового вводу/виводу з завдання 1.7 ґ.

Завдання 1.11. Побудувати блок-схему виконання заданого оператора. Скласти таблицю пріоритетів операцій, що використовуються. Транслювати конкретний приклад в ПОЛІЗ:

а) For l:=5 to b+c step 3 ¶ d:=d+l ¶ next

б) If a+3 < b-1 then a := b * c – 3 else b := a – c;

в) Do a := b to 15 by -c; k := k + a;

г) If a + b > 0 and c = k or b < 0 then p := p * 15 ¶

ґ) Do a= 50 by -3 to c while (d < 5) ; d := d * a; a ≔ a + 1 end ;

д) If a < c+d or a > 5 and c = 0 goto M1 ;

е) Do while (a > 0 or a + b < -7 and b <> 0); a := a – 1; b ≔ b + a^2 end ;

є) If a < b/c then a := (a+50)*c ¶

ж) For(i ≔ 0; i < (n-1)/2; i ≔ i+2) {a ≔ a + (n – i); }

з) For i = 1, x, x^2, y, y^2, z, z^2 { s ≔ s+i; write (i, i^3) }

и) For i = x1 to x1*2, x1*2 + x2 to x1*2 + x2*2 do write (i) enddo

і) For i = x1 by 2 to x1*2, x1*2 + x2 by 3 to x1*2 + x2*2 do write (i);

ї) For i = 1, 2, 2^2 to 2^3, 2^4{ s ≔ s+i; write (i, s) }

й) For i = 1, 2 to 2^s by 2, 3 to 3^s by 3{ s ≔ s+i; write (i, s) }

к) For j = 1, 2 to 2^s, 3 to 3^s by 3, 5 to 5^s{ s ≔ s+i; write (i, s) }

л) For i := 5 – c to step 3 ¶ For j := 0 to c step 2 write(i, j) ¶ next ¶ next

м) If a < b then if b < c then a := b * c – 3 else a := b / c else b := a – c;

н) Do a := b to d by -c; if a > 0 then k := k + a;

о) If a + b > 0 then If c + b > 0 then p := p * (a + b) / (c + b) ¶

п) Do a= 50 by -3 to c while (d < 5) ; if a <> 0 then d := d * a; s ≔ s + a end;

р) Do a := b to d by –c begin If a < c + d goto M1; s := s + a end;

с) For(i ≔ 0; i < (n-1)/2; i ≔ i+2) { For(j ≔ 0; j < (n-1)/2; j ≔ j+2) {write(j);)}; write(i)}

т) For i = 1, x, x^2, y, y^2, z, z^2 { s ≔ s+i; write (i, i^3) }

у) For i = x1 to x1*2, x1*2 + x2 to x1*2 + x2*2 do write (i) enddo

ф) For i = x1 by 2 to x1*2, x1*2 + x2 by 3 to x1*2 + x2*2 do write (i);

х) For i = 1, 2, 2^2 to 2^3, 2^4{ s ≔ s+i; write (i, s) }

ц) For i = 1, 2 to 2^s by 2, 3 to 3^s by 3{ s ≔ s+i; write (i, s) }

ч) For j = 1, 2 to 2^s, 3 to 3^s by 3, 5 to 5^s{ s ≔ s+i; write (i, s) }

Завдання 1.12. Записати вирази завдання 1.10 у вигляді тріад.

Завдання 1.13. Записати вирази завдання 1.10 у вигляді тетрад.

Завдання 1.14. Для граматик простого передування, побудованих в завданні 1.8, визначити семантичні підпрограми для побудови тетрад.

Завдання 1.15. Перетворити мови завдання 1.8 для рекурсивного спуску, написати відповідні нетерміналам підпрограми для побудови тетрад.

РОЗДІЛ 2. ОРГАНІЗАЦІЯ ПАМ'ЯТІ КОМПІЛЯТОРА

2.1. Методи організації таблиць компілятора

2.1.1. Неврегульовані та впорядковані таблиці

Блок генерації коду вимагає наявності даних про використовувані в програмі ідентифікатори. Ці дані зберігаються в таблиці ідентифікаторів (або символів) [22; 23; 24].

В процесі компіляції новий елемент додається в таблицю один раз, але пошук ведеться кожний раз, коли в коді зустрічається ідентифікатор. Процедура пошуку потребує багато часу при компіляції, тому для підвищення швидкості компіляції важливо вибрати таку організацію таблиць, яка зможе забезпечити ефективний пошук.

Ефективність роботи з таблицями визначатимемо за числом порівнянь, які треба виконати, щоб знайти даний символ, оскільки це число впливає на час пошуку. Ефективність залежить не тільки від методу пошуку, але і від коефіцієнта завантаження таблиці, що представляє відношення поточного числа елементів n до максимально можливого числа елементів таблиці N.

Простий спосіб організації таблиць полягає в тому, щоб додавати елементи до таблиці в порядку їх надходження. Такі таблиці називають **неврегульованими**. Пошук в цьому випадку вимагає порівняння з кожним елементом таблиці, поки не буде знайдено потрібний.

Для неврегульованої таблиці з n елементів в середньому при пошуку буде викуватися по $n/2$ порівнянь. Якщо n достатньо велике (20 або більше), цей спосіб неефективний.

Пошук може бути ефективнішим, якщо елементи таблиці **впорядковані** деяким чином. Упорядкування може відбуватися, наприклад, в лексикографіч-

ному (алфавітному) порядку. Наприклад, імена "A", "AB", "ABC", "AC", "BB" розташовані в зростаючому лексикографічному порядку. Ефективним методом пошуку в упорядкованому списку є бінарний пошук, який ще називається дихотомією, логарифмічним пошуком та методом половинного ділення.

Ідея **бінарного пошуку** базується на поділі таблиці навпіл на кожному кроці. Символ **S**, який треба знайти, порівнюється з серединним елементом, номер якого в таблиці $(n+1)/2$. Якщо серединний елемент не дорівнює шуканому символу, пошук продовжується. Тут можливі два варіанти: якщо **S** менше серединного елемента, пошук продовжується серед елементів першої половини таблиці від 1 до $(n+1)/2-1$; інакше, якщо **S** більше серединного елемента, пошук продовжується по другій частині таблиці від $((n+1)/2+1)$-го до n-го елемента. Процес повторюється над відповідною частиною таблиці, доки черговий серединний елемент не буде дорівнювати шуканому **S**. Оскільки на кожному кроці число елементів, які можуть містити **S** скорочується вдвічі, то максимальна кількість порівнянь дорівнює $1+\log_2 n$.

Якщо $n=2$, для пошуку довільного елемента таблиці необхідно максимум 2 порівняння, якщо $n=4$, то 3; якщо $n=8$, то 4. Для $n=128$ потрібно максимум 8 порівнянь, тоді як для пошуку по неврегульованій таблиці буде потрібно в середньому 64 порівняння.

Процедура бінарного пошуку може бути реалізована ітеративно чи рекурсивно. Наведемо приклад ітеративної реалізації бінарного пошуку у вигляді функції BinSearchI (Рисунок 2.1), що отримує масив елементів T (реалізований спеціальним класом Table) та шукане значення S типу елементів таблиці (Element) і повертає номер шуканого елемента в масиві. Рекурсивну реалізацію даного алгоритму покажемо у вигляді функції BinSearchR (Рисунок 2.2). Дана функція приймає масив елементів T, шукане значення S, а також номери початку та кінця частини таблиці, в якій необхідно шукати S. Оскільки на початку пошук виконується по всій таблиці, при виклику функції BinSearchR в якості останніх двох параметрів слід передати номер початкового та кінцевого елементів таблиці T.

```
function BinSearchI(T : Table; S : Element) : Integer;
  /* low – номер початку блока     high – номер кінця блока
  M – номер шуканого елемента      N – розмір таблиці T
  K – номер серединного елемента */
begin
  low ≔ 0;//на початку блоком є вся таблиця від початкового,0-го
  high ≔ T.high(); // до останнього елемента
  M ≔ -1;//елемент не знайдено: індекс поза межами таблиці
  while (low ≤ high and M = -1)
    begin
      K ≔ (high + low) / 2;
      //порівняння шуканого з серединним елементом
      if (S ≠ T[K]) then
        if (S<T[K]) then high ≔ K – 1
        else low ≔ K + 1
      else M ≔ K;
    end;
end; // of BinSearchI
```

Рисунок 2.1 – Ітеративна функція бінарного пошуку

```
function BinSearchR(  T : Table; S : Element;
                      low, high : Integer) : Integer;
  /* low – номер початку блока     high – номер кінця блока
  M – номер шуканого елемента      N – розмір таблиці T
  K – номер серединного елемента */
begin
  if (high < low) then
    Result ≔ -1 // ознака, що елемент не знайдено
  else
    begin
      K ≔ (high + low) / 2;
// порівняння шуканого з серединним елементом
      if (T[K] = S) then Result ≔ K // елемент знайдено
      else if (S < T[k]) then BinSearchR(T, S, low, K – 1)
      else /*тобто S > T[k]*/BinSearchR(T, S, K + 1, high);
    end;
end; // of BinSearchI
```

Рисунок 2.2 – Рекурсивна функція бінарного пошуку

Оскільки таблиця має бути впорядкованою, якщо заповнення передує пошуку, впорядкування можна виконати після заповнення. Якщо заповнення і

пошук чергуються, то кожний новий елемент можна додавати в таблицю методом впорядкованих вставлень. Нехай необхідно додати в таблицю елемент S, це можна реалізувати наступним чином.

Алгоритм впорядкованих вставлень

1) Методом бінарного пошуку знаходиться таке K, що $T_K < S < T_{K+1}$.
2) Виконується зсув елементів таблиці, що стоять після K, вправо. Останній елемент під номером N переміщується в позицію $N+1$; $(N-1)$-ий в позицію N; … ; $(K+1)$-ий елемент – в позицію $K+2$.
3) S заноситься в позицію $K+1$ (яка звільнилася після п. 2).

2.1.2. Перемішані таблиці. Хеш-адресація

Хеш-адресація – це найбільш ефективний і широковживаний в компіляторах метод роботи з таблицями символів (ідентифікаторів).

Метод полягає в перетворенні символу (ідентифікатора) на індекс елемента в таблиці. Індекс отримується в результаті операції, що називається **хешуванням** символу. Хешування полягає у виконанні над символом (і, можливо, над його довжиною) деяких простих арифметичних або логічних операцій.

Механізм хешування базується на **функціях хешування** (які ще називають **хеш-функціями** або **функціями розстановки**) h, таблиці хешування і таблиці даних (Рисунок 2.3). Таблиця даних може збігатися з таблицею хешування, тоді загальна таблиця має вигляд:

Ім'я	Значення

Функція розстановки – це фактично набір функцій $h_0, h_1, ..., h_m$, кожна з яких відображає набір об'єктів на множину цілих невід'ємних чисел $0,1,...,N-1$. Функцію $h_0(S)$ називають **первинною функцією розстановки**. Коли зустрічається об'єкт S, функція $h(S)$ вказує на адресу об'єкта в таблиці, якщо він зустрічався

раніше, або адресу порожньої комірки таблиці, в яку треба записати об'єкт, якщо він зустрічається вперше.

Рисунок 2.3 – Схема пам'яті, що використовує хешування (розстановку)

Обчислення адреси в таблиці розстановки

Вхід: об'єкт *S*, функція розстановки, що складається з послідовності функцій, кожна з яких відображає множину об'єктів на множину цілих чисел, таблиця розстановки з *N* комірками.

Вихід: адреса розстановки $h(S)$ і вказівка, чи зустрічався об'єкт S раніше.

Алгоритм

1) Обчислюємо по черзі функції $h_0(S), h_1(S), ..., h_m(S)$, виконуючи крок 2 до тих пір, поки не зникне «конфлікт». Якщо $h_m(S)$ дає конфлікт, необхідно зупинитися та повідомити про невдачу.
2) Обчислюємо $h_i(S)$, на початку $i = 0$:
   а) якщо комірка під номером $h_i(S)$ в таблиці розстановки порожня, вважаємо $h(S) = h_i(S)$, повідомляємо, що об'єкт S раніше не зустрічався, і зупиняємося;

b) якщо комірка під номером $h_i(S)$ не порожня, перевіряємо поле її імені. Якщо ім'ям є S, вважаємо $h(S)=h_i(S)$, повідомляємо, що об'єкт уже зустрічався, і зупиняємося. Якщо ім'я в комірці не збігається з S, значить виник **конфлікт**, повторюємо крок 2 для обчислення іншої адреси, тобто $h_{i+1}(S)$.

Обчислення кожної функції $h_i(S)$ і перевірка вмісту таблиці за цією адресою називається **пробій таблиці**.

Коли таблиця розстановки мало заповнена, конфлікти трапляються рідко, і значення $h(S)$ для нового об'єкта можна обчислити швидко (часто просто $h_0(S)$). По мірі заповнення таблиці конфлікти стають частішими. Однак можна сконструювати хеш-таблиці таким чином, що за характеристиками ефективності пошуку вони будуть перевершувати двійковий пошук.

Приклад 2.1

Нехай розмір хеш-таблиці $N=10$, а об'єктами є довільні ланцюжки латинських букв.

Визначимо функції $h(S)=\text{CODE}(S)$ як суму «порядкових номерів» кожної букви ланцюжка S. Вважаємо, що буква "A" має порядковий номер 1, "B" – 2 і так далі. Визначимо $h_i(S)$ для $0\le i\le 9$ як

$$h_i(S)=(\text{CODE}(S)+i)\bmod 10,$$

де $a \bmod b$ – це залишок від ділення a на b.

Необхідно внести в таблицю розстановки ланцюжки "A", "W" та "EF".

Розв'язання. Спершу вносимо об'єкт "A", для цього обчислюємо первинну функцію розстановки $h_0(\text{"A"})=(1+0)\bmod 10=1$. Оскільки в комірці під номером 1 порожньо, вносимо "A" в позицію 1.

Тепер внесемо ланцюжок "W", обчислимо $h_0(\text{"W"})=(23+0)\bmod 10=3$, у третій комірці порожньо – вносимо туди "W".

Наступний ланцюжок – "EF". Обчислюємо для нього функцію розстановки $h_0(\text{"W"}) = (5+6+0) \bmod 10 = 1$. Оскільки позиція номер 1 уже зайнята, обчислюємо наступну функцію $h_1(\text{"W"}) = (5+6+1) \bmod 10 = 2$. Комірка під номером 2 порожня, отже можна записати в неї даний ланцюжок "EF" (Рисунок 2.4).

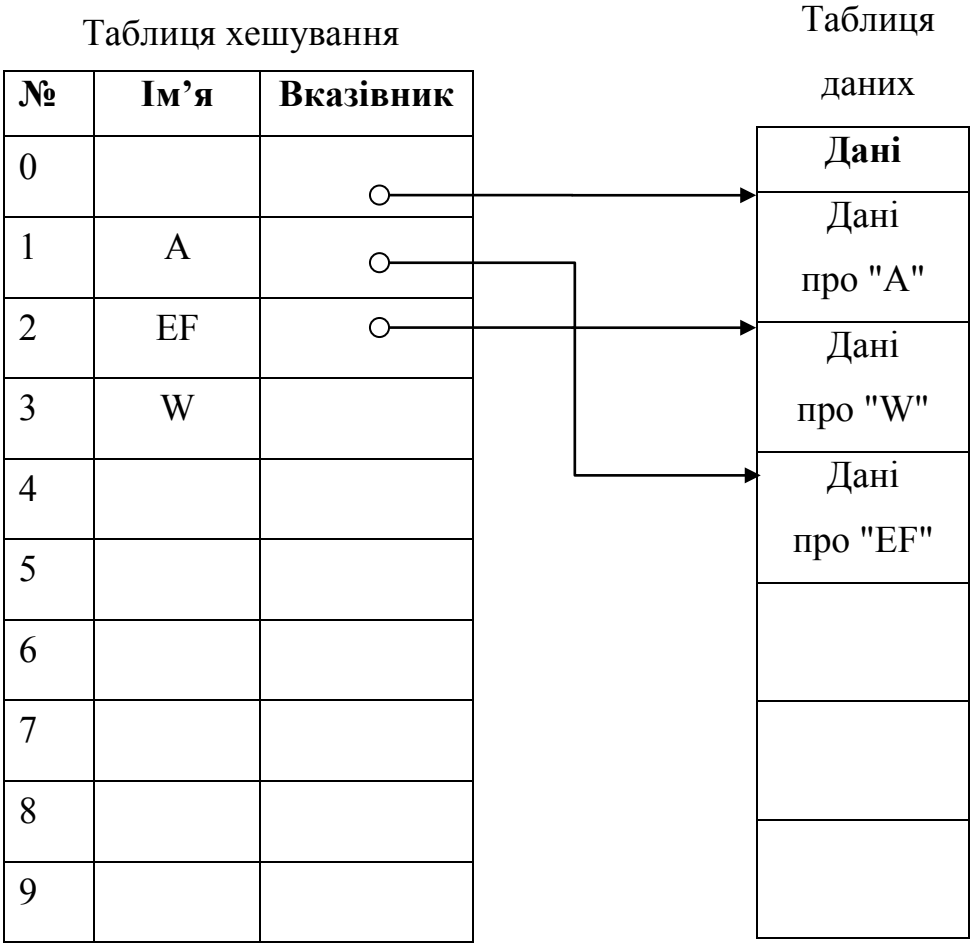


Рисунок 2.4 – Вміст хеш-таблиці

Нехай тепер треба з'ясувати, чи є в таблиці об'єкт "HX".

Знаходимо $h_0(\text{"HX"}) = (8+24+0) \bmod 10 = 32 \bmod 10 = 2$. Перевіряємо комірку номер 2: вона не пуста, але ім'я в комірці ("W") не збігається з шуканим ("HX"), значить виник конфлікт.

Тоді обчислюємо наступну функцію $h_1(\text{"HX"}) = (8+24+1) \bmod 10 = 3$. Перевіряємо позицію номер 3 – знову конфлікт ("EF" ≠"HX").

Нарешті обчислюємо $h_2(\text{"HX"}) = (8+24+2) \bmod 10 = 4$. В комірці номер 4 порожньо, приходимо до висновку, що об'єкта "HX" в таблиці немає.

Функції розстановки

Бажано користуватися такою первинною функцією розстановки, яка б розподіляла об'єкти рівномірно по всій таблиці хешування. Треба уникати функцій, які відображають не на всю множину позицій або вибирають одні позиції частіше за інші.

Найчастіше первинні хеш-функції обчислюються, як описано нижче.

Якщо S займає в машині декілька слів, то на першому кроці хешування по S формується одне машинне слово S'. Як правило S' обчислюється складанням усіх слів за допомогою звичайного або порозрядного додавання.

На другому кроці за S' обчислюється остаточний індекс одним з наступних способів:

1) помножити S' на себе і використовувати n середніх бітів як значення функції хешування (таблиця містить 2^n елементів). Оскільки n середніх бітів залежать від усіх бітів S', цей метод дає добрі результати;
2) якщо в таблиці 2^n елементів, розбити S' на n частин, скласти їх, використовувати n правих бітів результату.
3) S' розділити на розмір таблиці і взяти за індекс залишок від ділення.

У компіляторі PL/1 рівня F використовується наступна хеш-функція:

1) складаються послідовні частини ідентифікатора, що містять по 4 літери в один 4-байтний регістр;
2) результат ділиться на 211, залишок записується в R;
3) значення $2R$ береться як індекс в хеш-таблиці з 211 вказівників (кожний вказівник має довжину 2 байти).

Зручно було б, якби $m=n-1$, і щоб $h_i \neq h_j$ для будь-яких i, j, оскільки бажано знайти в таблиці порожню позицію, якщо вона є.

Розглянемо методи вирішення конфліктів; тобто побудови додаткових функцій $h_1(S), h_2(S), ..., h_m(S)$, цей процес називається **рехешуванням**. Метод вирішення конфліктів впливає на ефективність всієї системи розподілу пам'яті.

Найпростішим набором хеш-функцій є

$$h_i(S) = (h_0(S) + i) \bmod N, \forall\ 1 \le i \le N-1.$$

Перевагою такого методу є простота. Однак, якщо виник конфлікт, то зайняті позиції мають тенденцію скупчуватися, що вважається негативним ефектом.

Більш ефективний метод отримання вторинних хеш-адрес полягає у виборі таких функцій рехешування:

$$h_i(S) = (h_0(S) + r_i) \bmod N, \forall\ 1 \le i \le N-1,$$

де r_i – псевдовипадкове число. Для обчислення r_i достатньо використовувати генератор випадкових чисел, що видає всі числа в діапазоні від 1 до $N-1$ по одному разу. Кожного разу, коли використовуються вторинні функції, генератор встановлюється в один і той самий стан, тобто він кожного разу генерує одну й ту саму послідовність чисел.

В якості вторинних функції можна взяти також наступні:

$$h_i(S) = [i(h_0(S) + 1)] \bmod N;$$
$$h_i(S) = [h_0(S) + a * i^2 + b * i] \bmod N,$$

де a та b – деякі константи.

2.1.3. Ефективність методів рехешування. Ланцюжки.

Лінійне рехешування

Розглянутий раніше найпростіший набір хеш-функцій відповідає способу лінійного рехешування:

$$h_i = (h_0(S) + i) \bmod N,\ 1 \le i \le N-1.$$

Для демонстрації недоліків даного способу рехешування розглянемо приклад.

Приклад 2.2

Припустимо, що символи $S1$ і $S2$ були захешовані та записані в позиції таблиці під номером 2 і 4, відповідно (Рисунок 2.5 а).

№	Ім'я	Вказівник
0		
1		
2	$S1$	
3		
4	$S2$	
5		
6		
7		

а)

№	Ім'я	Вказівник
0		
1		
2	$S1$	
3	$S3$	
4	$S2$	
5		
6		
7		

б)

№	Ім'я	Вказівник
0		
1		
2	$S1$	
3	$S3$	
4	$S2$	
5	$S4$	
6		
7		

в)

Рисунок 2.5

Тепер припустимо, що символ $S3$ за первинною хеш-функцією теж посилається на позицію 2. Виникає конфлікт. Обчислюємо першу допоміжну функцію $h_1(S)=3$ і заносимо $S3$ в комірку №3 (Рисунок 2.5 б).

Нехай після цього до таблиці необхідно занести символ $S4$, первинна хеш-функція якого також дорівнює 2. Виникає послідовно три конфлікти з $S1$, $S3$ і $S2$, оскільки $h_0(S4)=2$, $h_1(S4)=3$, $h_2(S4)=4$, і тільки третя допоміжна хеш-функція вказує на вільну позицію в таблиці $h_3(S4)=5$, символ $S4$ заноситься в п'яту комірку (Рисунок 2.5 в).

Низька ефективність даного способу рехешування добре помітна на даному прикладі. Після декількох конфліктів, вирішених таким способом, елементи скупчуються в одному місці, утворюючи в таблиці послідовності заповнених елементів. Якщо досягнута позиція $N-1$, елементи переходять на позицію 0. Якщо $h_0(S)$ викликає конфлікт, то вірогідність того, що $h_1(S)$ викличе конфлікт, – вище середнього.

Оцінка **середнього числа порівнянь** відповідає формулі:

$$E = \frac{\left(\frac{1-lf}{2}\right)}{1-lf}.$$

Таким чином залежність очікуваного числа порівнянь E від завантаження таблиці lf буде наступною (Таблиця 2.1).

Таблиця 2.1. Оцінка ефективності лінійного рехешування

lf	E
10%	1,06
50%	1,5
90%	5,5

Слід відзначити, що кількість порівнянь залежить не від розміру таблиці, а тільки від ступеня її заповнення.

У формулі лінійного рехешування замість i можна знайти кращі значення поправок. Але навіть при використанні i цей метод швидший за бінарний пошук. Припустимо є таблиця з 1024 елементів заповнена наполовину, тобто 512 елементів. У випадку бінарного пошуку очікується виконання в середньому $(1+\log_2 N) = 11$ порівнянь, а при лінійному рехешуванні лише 1,5.

Випадкове рехешування

Ще один спосіб рехешування базується на функціях, в яких замість r_i використовуються псевдовипадкові числа так само, як було описано в лінійному способі:

$$h_i(S) = (h_0(S) + r_i) \bmod N, \quad 1 \le i \le N-1$$

При використанні даного способу проблема скупчення не виникає. Якщо розмір таблиці дорівнює степені двійки ($N=2^p$), то ефективним є використання генератора псевдовипадкових чисел, що був запропонований Морісом.

Алгоритм роботи генератора псевдовипадкових чисел Моріса

1) Встановити початкове значення $R=1$.
2) Обчислювати кожне r_i таким чином:
   a) встановити $R=R*5$;
   b) $R=R \bmod (4N)$, тобто взяти $p+2$ молодших розряди R і помістити в R;
   c) $r=\text{int}(R/4)$, тобто виконати порозрядний зсув над величиною R на 2 розряди вправо та занести результат в r, що і буде остаточним псевдовипадковим числом r_i.

Найважливіша властивість цього способу рехешування полягає в тому, що всі поправки r_i різні. Вони покривають усі адреси в таблиці.

Для даного методу достатньо точне наближення очікуваної кількості порівнянь дає формула:

$$E=\left(\frac{1}{lf}\right)\log(1-lf).$$

Залежність очікуваного числа порівнянь від завантаження таблиці буде наступною (Таблиця 2.2).

Таблиця 2.2. Оцінка ефективності випадкового рехешування

lf	E
10%	1,05
50%	1,39
90%	2,56

При великому рівні заповнення таблиці для аналізу кількості порівнянь використовується інша формула:

$$E = \frac{1}{1 - \frac{n}{N}}.$$

Приклад 2.3

Розглянемо процес генерування псевдовипадкових чисел для таблиці розміром $N = 8 = 2^3$. Обчислимо $4N = 32$, тоді за алгоритмом Моріса будуть згенеровані наступні поправки r_i (Таблиця 2.3).

Таблиця 2.3

№ поправки	1	2	3	4	5	6	7	8
R	1	5	25	29	17	21	9	13
$R = 5R$	5	25	125	145	85	105	45	65
$R = R \bmod 4N$	5	25	29	17	21	9	13	1
$r = \text{int}(R/4)$	1	6	7	4	5	2	3	0

Квадратичне рехешування

При використанні для поправок квадратичної функції, хеш-функція має наступний вигляд:

$$h_i = \left[h_0(S) + ai^2 + bi + c\right] \bmod N.$$

Головна проблема даного способу полягає в тому, щоб забезпечити покриття значеннями $r_i = ai^2 + bi + c$ достатньо великої кількості елементів таблиці. Виявляється, що якщо розмір таблиці $N = 2^p$, то число елементів, що переглядаються, дуже мале.

Приклад 2.4

Розглянемо заповнення хеш-таблиці для $a = 1$, $b = 1$, $c = 0$ розмір таблиці $N = 8$. Припустимо $h_0(S) = 3$, обчислимо можливі $h_i(S)$ (Таблиця 2.4).

Таблиця 2.4

i	h_i
1	$(3+2) \bmod 8 = 5$
2	$(3+6) \bmod 8 = 1$
3	$(3+12) \bmod 8 = 7$
4	$(3+20) \bmod 8 = 7$
5	$(3+12) \bmod 8 = 1$
6	$(3+42) \bmod 8 = 5$
7	$(3+56) \bmod 8 = 3$

Вибрано тільки 3 нових адреси: 1,5,7. Якщо N – просте число, то кількість елементів, що перевіряються, складає половину таблиці. Цей метод не такий добрий, як випадкове хешування, але він вимагає меншого часу для обчислення r_i, ніж випадкове хешування.

Метод ланцюжків

Для реалізації методу ланцюжків необхідно мати наступні елементи:

- хеш-таблицю, елементи якої (вказівники), в початковий момент дорівнюють 0;
- спочатку порожню таблицю символів;
- вказівник на найближче вільне місце в таблиці символів (POINTFREE).

Елементи таблиці символів мають додаткове поле, яке може містити нуль або адресу іншого елемента таблиці символів.

Таким чином, початковий стан всієї системи наступний (Рисунок 2.6).

Додаткове поле кожного елемента використовується для того, щоб пов'язати в ланцюжок елементи, для яких хеш-функція видає однакові результати.

Рисунок 2.6 – Початковий стан таблиць методу ланцюжків

Нехай символ $S1$ має бути записаний в таблицю символів. Функція хешування видає деяку адресу, наприклад 4. Це адреса в хеш-таблиці. Якщо в хеш-таблиці за цією адресою знаходиться 0, то для запису елемента виконуються наступні дії:

1) внести елемент ($S1$ зі значенням 0) у позицію таблиці, на яку вказує POINTFREE;
2) занести вміст POINTFREE в 4-у комірку хеш-таблиці (оскільки 4 – результат функції хешування).
3) збільшити вказівник POINTFREE на 1.

Отримаємо (Рисунок 2.7).

Якщо надходять символи, для яких хеш-функція дає різні адреси хеш-таблиці, процедура запису аналогічна описаній.

Рисунок 2.7 – Таблиці методу ланцюжків після додавання першого елемента

Отже після запису ще трьох елементів, для яких по черзі було сформовано адреси хеш-таблиці: 1, 3, 6, – стан системи набув такого вигляду (Рисунок 2.8).

Рисунок 2.8 – Таблиці методу ланцюжків після додавання чотирьох елементів без виникнення конфліктів хеш-адрес

Тепер розглянемо ситуацію, коли хеш-функція формує вже зайняту адресу. Припустимо на вхід надходить символ $S5$ і хеш-функція формує для нього адресу хеш-таблиці 6. Вміст хеш-таблиці за цією адресою не дорівнює 0, там зберігається адреса в таблиці символів, за якою знаходиться елемент (його ім'я $S4$, в його додатковому полі 0). Для вирішення даного конфлікту слід виконати наступні дії:

1) внести до додаткового поля елемента, що містить $S4$, значення адреси наступного вільного рядка таблиці символів (POINTFREE);
2) в рядок таблиці символів за адресою (що міститься в POINTFREE) записати $S5$, його значення та 0 в додаткове поле .
3) збільшити вказівник POINTFREE на одиницю.

Після виконання описаних дій, система перейде в наступний стан (Рисунок 2.9).

Нехай після цього послідовно будуть занесені ще три елементи $S6$, $S7$, $S8$, які посилаються на адреси 4, 3 та 3, відповідно. Стан системи набуде вигляду (Рисунок 2.10).

Наприклад, тепер необхідно знайти значення елемента $S7$. За первинною хеш-функцією знаходимо його хеш-адресу: 3. За номером 3 в хеш-таблиці збе-

рігається адреса відповідного елемента в таблиці символів: 3. Перевіряємо 3-й елемент таблиці символів: його ім'я відмінне від шуканого $(S3 \neq S7)$, значить необхідно перейти за адресою, вказаною в додатковому полі цього елемента (там знаходиться адреса 7). Перевіряємо 7-й елемент таблиці символів: його ім'я збігається з шуканим – елемент знайдено.

Рисунок 2.9 – Таблиці методу ланцюжків після додавання п'ятого елемента, що має конфлікт адреси з четвертим

Рисунок 2.10

Відтворимо ситуацію, коли відбувається пошук елемента, якого немає в таблиці символів. Нехай елемент називається $S9$, і його первинна хеш-функція повертає 6. В хеш-таблиці знаходимо 6-й елемент, там міститься посилання на 4-й елемент таблиці символів. Оскільки ім'я 4-ого елемента таблиці символів не збігається з шуканим $S9$ $(S4 \neq S9)$, перевіряємо додаткове поле 4-ого елемента:

там знаходиться адреса 5-ого елемента. Перевіряємо 5-й елемент: його ім'я також не збігається з шуканим ($S5 \neq S9$). Оскільки в додатковому полі не міститься адреси на інший елемент, можна зробити висновок, що елемента $S9$ немає в таблиці.

Однією з переваг методу ланцюжків є те, що в разі заповнення таблиці символів, до неї можна завжди додати новий блок елементів (за умови використання динамічного розподілу пам'яті). Оскільки функція хешування дає адресу в хеш-таблиці, в якій міститься посилання на таблицю символів, що реалізується за допомогою вказівників, таке додаткове виділення блоків таблиці символів не вплине на метод доступу до елементів.

Таким чином, максимальна кількість елементів в таблиці символів при використанні методу ланцюжків необмежена, на відміну від методів рехешування.

В даному методі накладається обмеження лише на максимальне число адрес хеш-таблиці. Оскільки лише хеш-таблиця вимагає попереднього виділення пам'яті та встановлення значень за замовчуванням. Розмір хеш-таблиці зазвичай 100-300 елементів, а число елементів в таблиці символів набагато більше. Як тільки всі символи внесені до таблиці (наприклад, після лексичного аналізу), хеш-таблицю взагалі можна видалити, попередньо замінивши всі ідентифікатори на команди лексем з зазначеними адресами елементів в таблиці символів.

До недоліків можна віднести те, що метод ланцюжків вимагає одне додаткове машинне слово для кожного елемента, але в більшості випадків ця витрата пам'яті нівелюється іншими перевагами даного методу.

Якщо припустити випадковий розподіл елементів, то середня кількість порівнянь при пошуку (довжина пошуку) може бути знайдена за формулою:

$$E = 1 + \frac{n-1}{2N}.$$

Характерно, що навіть при заповненій на 100% таблиці, тобто при $n = N$, середня довжина пошуку не перевищує 1,5.

Розглянутий варіант реалізації називається **методом внутрішніх ланцюжків**. Існує варіант **зовнішніх ланцюжків**, коли таблиці системи організовані таким чином, що для кожного елемента, який викликав конфлікт адрес, створюється окрема ланка пам'яті, де міститься ім'я, значення («Знач.») та додаткове поле («Д.п.») для адреси наступного конфліктуючого елемента. Отже основна таблиця символів містить перелік голів (початкових елементів) однонаправлених списків елементів, що конфліктують.

За такої організації система, в яку додали той самий перелік символів, як в попередньому прикладі, що за умови використання методу внутрішніх ланцюжків мала такий вигляд (Рисунок 2.10), набуде дещо іншого вигляду (Рисунок 2.11).

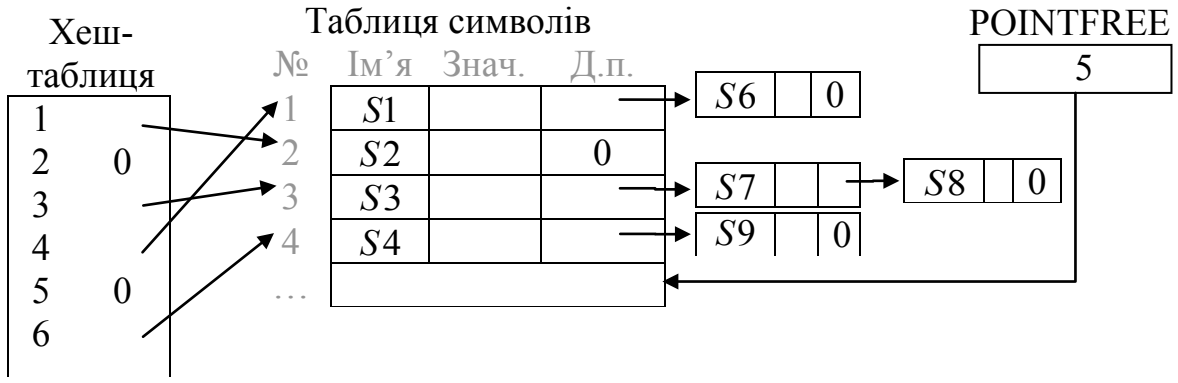


Рисунок 2.11 – Приклад організації пам'яті з використанням зовнішніх ланцюжків

Основною відмінністю методів внутрішніх та зовнішніх ланцюжків є розміщення елементів в пам'яті. В першому випадку усі елементи, в тому числі конфліктуючі, розміщуються в послідовному масиві, доступ до елементів якого виконується за номером. У другому випадку в такому масиві містяться лише початкові елементи списків конфліктуючих елементів, що є головами однонаправлених списків елементів, хеш-адреси яких збігаються. При використанні однонаправлених списків реальне розміщення елементів в пам'яті комп'ютера може бути непослідовним. Очевидно, для способу організації зовнішніх ланцюжків потребується послідовний масив меншого розміру, що краще підходить

за умови використання дрібно-сегментованої кучі (частини пам’яті комп’ютера для збереження динамічних змінних).

2.2. Розподіл пам'яті

2.2.1. Пам'ять для даних

Типи даних початкової програми мають бути відображені на типи даних машини. Для деяких типів ця відповідність буде одна до одного (ціле, дійсне число), для інших може знадобитися декілька машинних слів для опису типу.

Пам'ять для масивів

Елементи масиву або **вектора** розміщуються послідовно в порядку зростання (або спадання) адрес. Доступ до елементів масиву надається через номер елемента в масиві.

Для розміщення в пам’яті двовимірних масивів їх слід перетворити на одновимірні – **лінеаризувати,** – існує кілька способів лінеаризації масивів.

Традиційний спосіб – **лінійне відображення** передбачає збереження елементів по рядках в порядку зростання. Елементи масиву $A(1{:}M,1{:}N)$ будуть розташовані в порядку

$$A(1,1), A(1,2),\ldots A(1,N), A(2,1),\ldots A(2,N),\ldots A(M,N).$$

Тоді елемент $A(i,j)$ знаходиться в комірці з адресою

$$BASE + (i-1)*N + (j-1), \qquad (2.1)$$

де $BASE$ = ADDR($A(1,1)$) – адреса першого елемента масиву.

Формулу (2.1) зручніше подавати у вигляді

$$(BASE - N - 1) + (i*N + j). \qquad (2.2)$$

Перевагою формули (2.2) є те, що перший доданок буде незмінним для всіх (i, j), тому він обчислюється тільки один раз. Таким чином, для обчислення адреси $A(i, j)$ виконується одна операція множення та дві – додавання.

Другий спосіб полягає в тому, що для кожного рядка виділяються окремі (не пов'язані між собою) області даних, крім того необхідно мати вектор вказівників на ці області. Наприклад, опис масиву $A(1{:}M, 1{:}N)$ породжує наступну структуру (Рисунок 2.12).

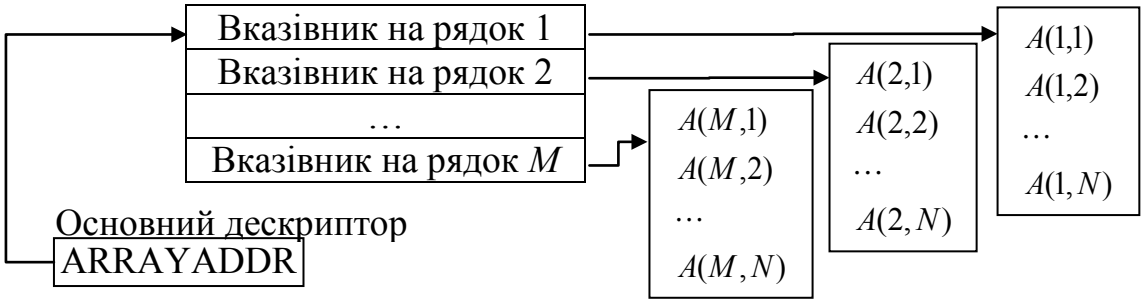


Рисунок 2.12 – Реалізація двовимірного масиву вектором вказівників на рядки

Вектор вказівників зберігається в тій області даних, з якою асоціюється масив (позначимо цю адресу (основний дескриптор) через ARRAYADDR), а власне елементи масиву зберігаються в окремій області даних. В такому випадку адреса елемента масиву $A(i, j)$ отримується за формулою:

$$(\text{ARRAYADDR} + i - 1) + (j - 1).$$

Переваги другого методу:

1) не треба використовувати операцію множення для визначення адреси елемента масиву;
2) не всі рядки повинні знаходитися в оперативній пам'яті одночасно;
3) необов'язкове послідовне розташування рядків у пам'яті.

Остання перевага особливо значуща у випадку дуже великих масивів, які необхідно розташовувати в пам'яті, що поділена на порівняно дрібні сегменти. Якщо обсяг масиву перевищує обсяг сегменту, застосувати перший спосіб взагалі неможливо, оскільки масив не вміститься в жодний з сегментів. У той са-

мий час при використанні другого методу рядки масиву можуть розміщуватися в різних сегментах пам'яті. Також ця перевага відчутна при частково зайнятій пам'яті машини, адже для розміщення масиву першим способом необхідно знайти в пам'яті послідовно розміщені $(M*N)$ незайнятих комірок, що може бути складною задачею при великих значеннях M та N.

Для розміщення в пам'яті багатовимірного масиву можна виконувати лінійне відображення аналогічне першому способу розміщення двовимірного масиву (в якому швидше змінюються правіші індекси).

Визначимо, як обчислюється адреса елемента $A(i, j, k, \ldots, l, m)$ для багатовимірного масиву $A(L_1 : U_1, L_2 : U_2, \ldots L_n : U_n)$.

Введемо позначення:

$$\begin{aligned} d_1 &= U_1 + L_1 + 1 \\ d_2 &= U_2 + L_2 + 1 \\ &\ldots \\ d_n &= U_n + L_n + 1 \end{aligned}$$

тобто d_η – кількість різних значень індексу η-го виміру масиву.

Тоді адреса шуканого елемента визначається за формулою:

$$\begin{aligned} BASE + (i + L_1) * d_2 * d_3 * \ldots * d_n + (j + L_2) * d_3 * \ldots * d_n + \\ + (k + L_3) * d_4 * \ldots * d_n + \ldots (l + L_{n-1}) * d_n + m - L_n, \end{aligned} \qquad (2.3)$$

де $BASE = \text{ADDR}(A(L_1, L_2, \ldots L_n))$ – адреса першого елемента масиву.

Виконавши множення, розділимо отриманий вираз (2.3) на постійну $(CONSPART)$ і змінну частини $(VARPART)$:

$$CONSPART = BASE - (\ldots((L_1 * d_2 + L_2) * d_3 + L_3) * d_4 + \ldots + L_{n-1}) * d_n + L_n),$$

$$VARPART = (\ldots((i * d_2) + j) * d_3 + \ldots + l) * d_n + m.$$

$CONSPART$ обчислюється один раз, вона залежить лише від розташування масиву та його розмірів, а $VARPART$ залежить від індексів елемента.

Обчислення $VARPART$ можна виконувати ітеративно:

$$VARPART := перший\ індекс(i)$$
$$VARPART := VARPART * d_2 + другий\ індекс(j)$$
$$VARPART := VARPART * d_3 + третій\ індекс(k)$$
$$\ldots$$
$$VARPART := VARPART * d_n + n-ий\ індекс(m)$$

Для виділення пам'яті під такий багатовимірний масив використовується деякий інформаційний вектор. Структура вектора відома, й пам'ять під нього може бути відведена в процесі трансляції. Пам'ять же під власне елементи масиву може бути відведена або в процесі трансляції, якщо верхня і нижня границі масиву відомі на час трансляції, або під час виконання програми після входу в блок, де оголошується масив.

Інформаційний вектор для масиву $A(L_1:U_1, L_2:U_2, \ldots L_n:U_n)$ має наступний вигляд (Таблиця 2.5).

Таблиця 2.5. Інформаційний вектор багатовимірного масиву

L_1	U_1	d_1
L_2	U_2	d_2
.	.	.
L_n	U_n	d_n
n	*CONSPART*	
BASE		

З вище приведених формул видно, що обчислення адреси елемента багатовимірного масиву може потребувати багато часу, оскільки при цьому повинні виконуватися операції додавання та множення, кількість яких пропорційна розмірності масиву.

Операцію множення можна виключити аналогічно другому методу подання двовимірних масивів, якщо застосовувати для адресації **вектори Айліффа** [25]. Для масиву будь-якої розмірності формується набір дескрипторів: основного та декілька рівнів допоміжних дескрипторів, що називаються векторами Айліффа (Рисунок 2.13).

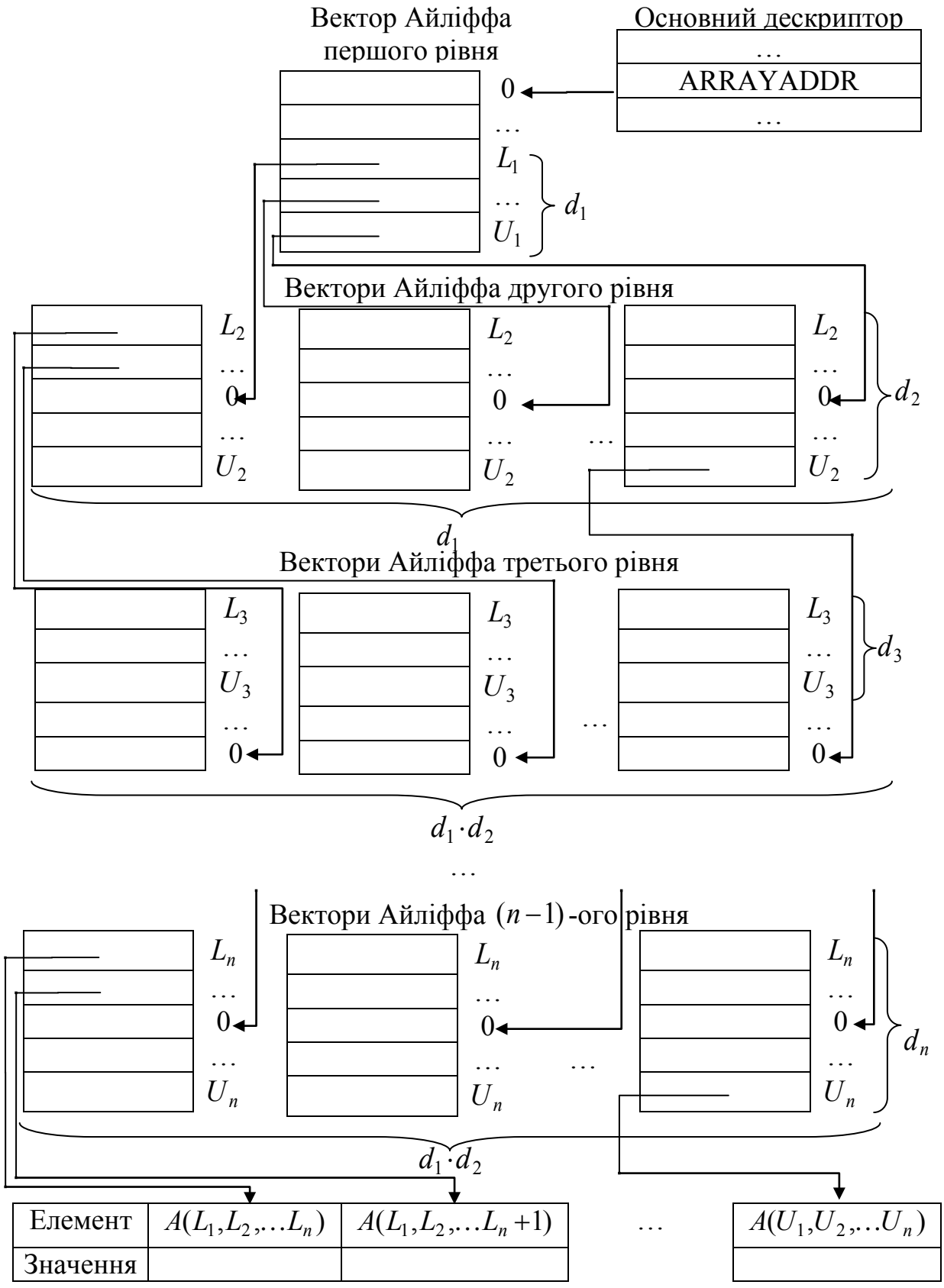


Рисунок 2.13 – Подання багатовимірних масивів методом Айліффа

Основний дескриптор масиву зберігає вказівник вектора Айліффа першого рівня. Кожний вектор Айліффа певного рівня містить вказівники на

нульові елементи векторів Айліффа наступного, нижчого рівня, а вектори Айліффа найнижчого рівня містять вказівники на реальні адреси, де розміщуються, власне, елементи массиву. Зверніть увагу, що нульові елементи можуть опинитися між L_i та U_i (Рисунок 2.13 вектори другого рівня), перед L_i (Рисунок 2.13 вектори першого рівня) або після U_i (Рисунок 2.13 вектори третього рівня). Очевидно, якщо нульовий елемент знаходиться поза межами відрізка $[L_i;U_i]$, під масив виділяється більша кількість пам'яті, ніж потрібно для збереження d_i кількості елементів.

При такій організації до довільного елемента $A(j_1,j_2,...,j_n)$ багатовимірного масиву можна звернутися, пройшовши по ланцюжку вказівників від основного дескриптора через відповідні елементи векторів Айліффа всіх рівнів.

Зі схематичного зображення (Рисунок 2.13) видно, що метод Айліффа, збільшуючи швидкість доступу до елементів масиву, призводить в той самий час до збільшення сумарного об'єму пам'яті, потрібного для розміщення масиву. В цьому полягає основний недолік представлення масивів з використанням векторів Айліффа.

Приклад 2.5

Нехай необхідно розмістити в пам'яті тривимірний масив $B[4:5,-1:1,0:1]$, його розмірність – 2*3*2. Припустимо, що елементи масиву з індексами *i*, *j*, *k* були розміщені в пам'яті послідовно, починаючи з нульової адреси (Рисунок 2.14).

№	0	1	2	3	4	5	6	7	8	9	10	11
i	0	0	0	0	0	0	1	1	1	1	1	1
j	0	0	1	1	2	2	0	0	1	1	2	2
k	0	1	0	1	0	1	0	1	0	1	0	1

Рисунок 2.14 – Лінійне відображення масиву $B[4:5,-1:1,0:1]$

Тоді значення адрес, що містяться в елементах векторів Айліффа можуть бути наступними (Таблиця 2.6).

Таблиця 2.6. Вектори Айліффа для масиву $B[4:5,-1:1,0:1]$

Вектори другого рівня		Вектор першого рівня
Перший $B21$ (Addr = 100)	Другий $B22$ (Addr = 150)	$B1$
-1 – 0 0 – 4 1 – 8	-1 – 12 0 – 16 1 – 20	4 – 100 5 – 150

Схематично фізична структура даного тривимірного масиву може бути зображена, як показано нижче (Рисунок 2.15).

Доступ до (i, j, k) -го елемента здійснюється таким чином:

Addr = (B1[i])[j] + k

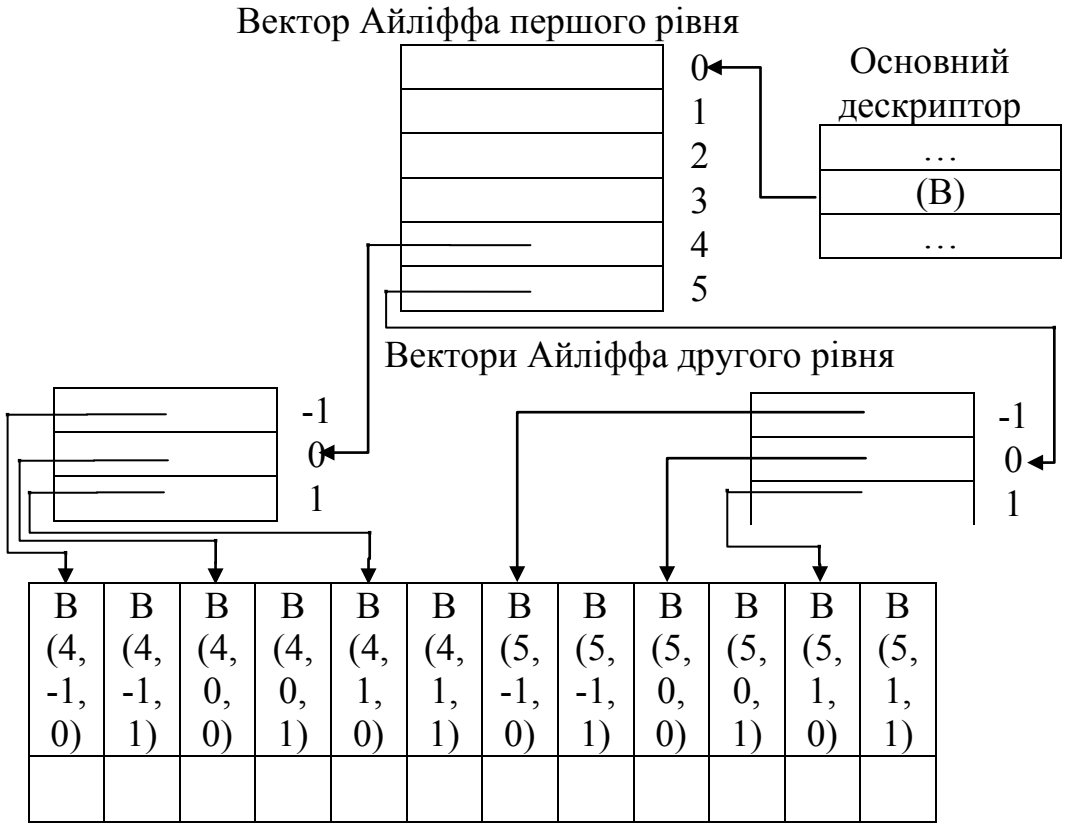


Рисунок 2.15 – Фізична структура масиву $B[4:5,-1:1,0:1]$

Приклад 2.6

Наведемо приклад подання тривимірного масиву C розмірністю 2*3*4, усі індекси якого починаються з нуля. Нехай елементи цього масиву були розміщені в пам’яті послідовно, починаючи з нульової адреси (Рисунок 2.16).

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23

0 0 0 0 0 0 0 0 0 0  0  0  1  1  1  1  1  1  1  1  1  1  1  1
0 0 0 0 1 1 1 1 2 2  2  2  0  0  0  0  1  1  1  1  2  2  2  2
0 1 2 3 0 1 2 3 0 1  2  3  0  1  2  3  0  1  2  3  0  1  2  3
```

Рисунок 2.16 – Лінійне відображення масиву *B*[4:5,–1:1,0:1]

Тоді значення адрес, що містяться в елементах векторів Айліффа, за умови що перший вектор другого рівня буде розміщений за адресою 100, а другий – 150, можуть бути настуними (Таблиця 2.7).

Таблиця 2.7. Вектори Айліффа для масиву розмірністю

Вектори другого рівня		Вектор першого рівня
Перший *C*21 (Addr = 100)	Другий *C*22 (Addr = 150)	*C*1
0 – 0 1 – 4 2 – 8	0 – 12 1 – 16 2 – 20	0 – 100 1 – 150

Розрізняють два способи розподілу пам'яті для розміщення даних: статичний та динамічний.

Статичний розподіл пам’яті полягає в призначенні адрес для розміщення даних в процесі трансляції.

Динамічний розподіл полягає в призначенні адрес для розміщення даних в процесі виконання програми. З цією метою транслятор включає в об'єктну програму спеціальні команди, що забезпечують перерозподіл пам'яті в ході виконання програми.

В більшості мов програмування для статичного розподілу пам’яті використовується частина пам’яті, що називається **стеком**, а для динамічного – **куча** (англ. heap).

В мовах програмування код поділяється на блоки. Блоки визначають зону дії описаних в них об'єктів і призначені, перш за все, для економії пам'яті в про-

цесі виконання програми. Економія досягається за рахунок повторного використання частин пам'яті для збереження змінних, описаних в незалежних блоках.

Приклад 2.7

При виконанні програми (Рисунок 2.17) до моменту досягнення відповідної мітки, що позначає новий блок, розподіл пам'яті змінюється таким чином (Таблиця 2.8). Тут припускається, що на початку програмі було виділено пам’ять, що починається з адреси **k**.

```
begin
  real a, b;
  b := 0;
  m1:
    begin
      real c, d;
      b := b + 1;
      m2: …
  end m1;
  n1:
    begin
      real f, q;
      n2:
        if b = 1 then goto m1;
          …
  end n1;
end;
```

Рисунок 2.17

Таблиця 2.8. Приклад 2.7 розподіл пам’яті

Мітка / Комірка	m1 b = 0	m2 b = 1	n1 b = 1	n2 b = 1	m1 b = 1	m2 b = 2	n1 b = 2	n2 b = 2
k + 0	a	a	a	a	a	a	a	a
k + 1	b	b	b	b	b	b	b	b
k + 2		c		f		c		f
k + 3		d		q		d		q

Змінні **c**, **d** та **f**, **q** по черзі займають ті самі комірки пам’яті (№ k+2, № k+3).

Цю блокову структуру можна уявити у вигляді дерева (Рисунок 2.18). Зовнішньому блоку програми відповідає перший рівень дерева, на цьому рівні виділено пам'ять для двох змінних типу real. В середині зовнішнього визначено блок, позначений міткою m1, в ньому також визначено дві змінні типу real: c, d. В блоці m1 оголошено ще одну мітку на блок m2, але в цьому блоці змінні не оголошуються, тому пам'ять не виділяється. Після завершення блока m1 оголошено блок n1, тобто змінні, оголошені в цьому блоці (f, q), будуть розташовані на тому ж рівні, що і a та b. В середині блока n1 оголошено блок n2, але оскільки в ньому не описано змінних, третій рівень дерева не виділяється.

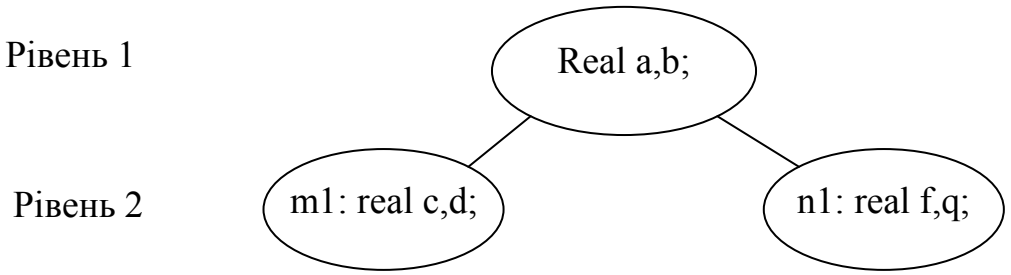


Рисунок 2.18

2.2.2. Статичний розподіл пам’яті

Статичний розподіл пам’яті можливий, якщо кількість даних, описаних в блоці, не змінюється при повторних входах в блок.

Статичний розподіл неможливий, якщо:

- застосовуються масиви із змінними межами, коли розміри масивів можуть бути визначені тільки при вході в блок;
- використовуються рекурсивні процедури, оскільки при їх виконанні потрібне копіювання області пам'яті виділеній процедурі, причому до виконання програми не можна визначити кількість копій.

Статичний розподіл проводиться в два етапи: на першому етапі виділяється секція пам'яті, відповідна кожному блоку коду, на другому – призначаються адреси змінним, описаним всередині блока.

Перший етап

Припустимо, програма має наступну блокову структуру (Рисунок 2.19). Цю структуру можна зобразити у вигляді дерева (Рисунок 2.20). За такої структури програми статична пам'ять може бути розподілена відповідно до наступної діаграми (Рисунок 2.21).

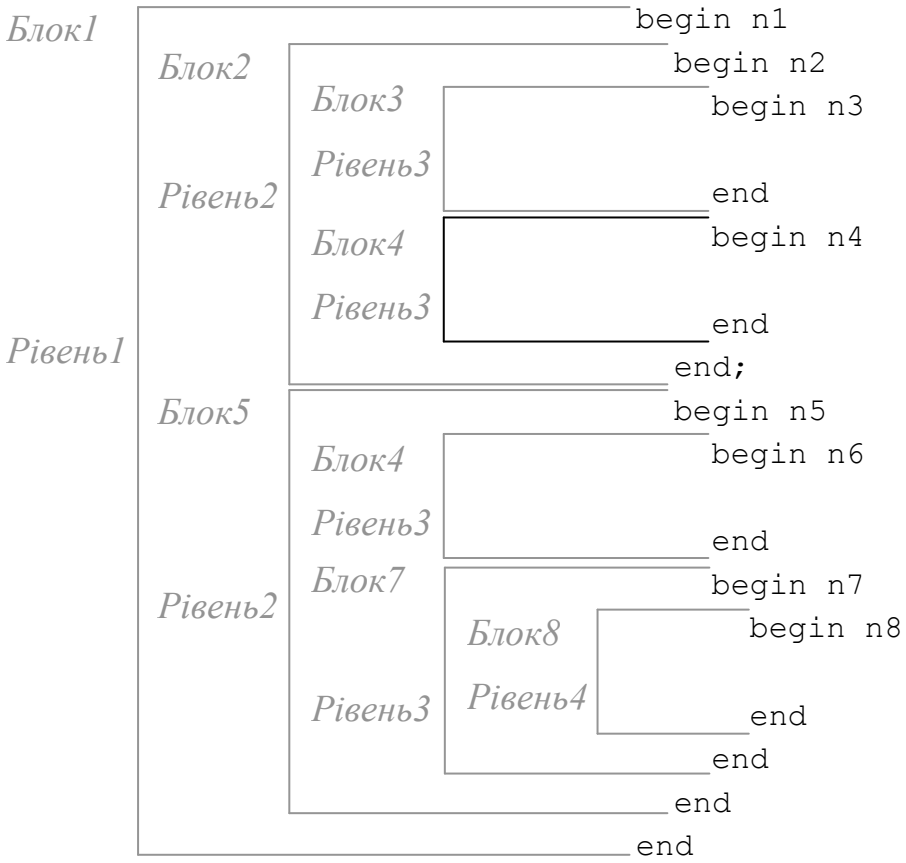


Рисунок 2.19 – Приклад структури програми

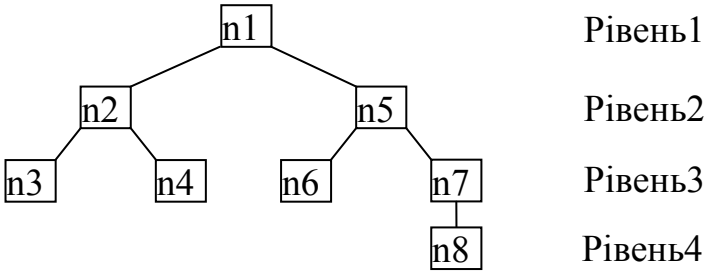


Рисунок 2.20 – Приклад структури програми у вигляді дерева

Для статичного розподілу пам'яті складається таблиця блоків. Кожний запис таблиці містить $РБ_i$ – рівень i-ого блока та ni – кількість комірок пам'яті

для розміщення даних блока. В N_i заноситься адреса елемента пам'яті, що передує першій комірці поля даних (Рисунок 2.22).

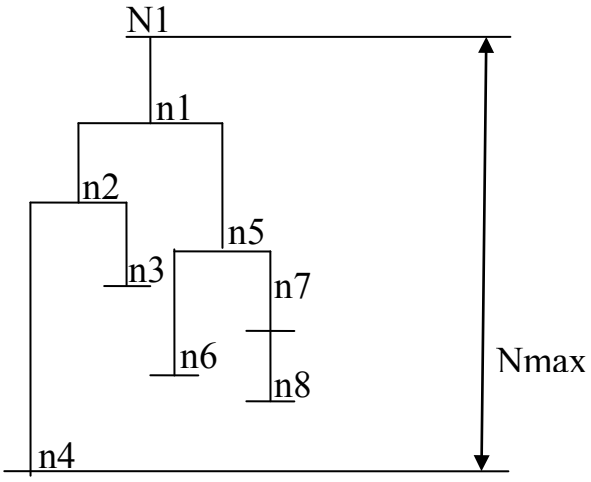


Рисунок 2.21 – Схема виділення пам’яті

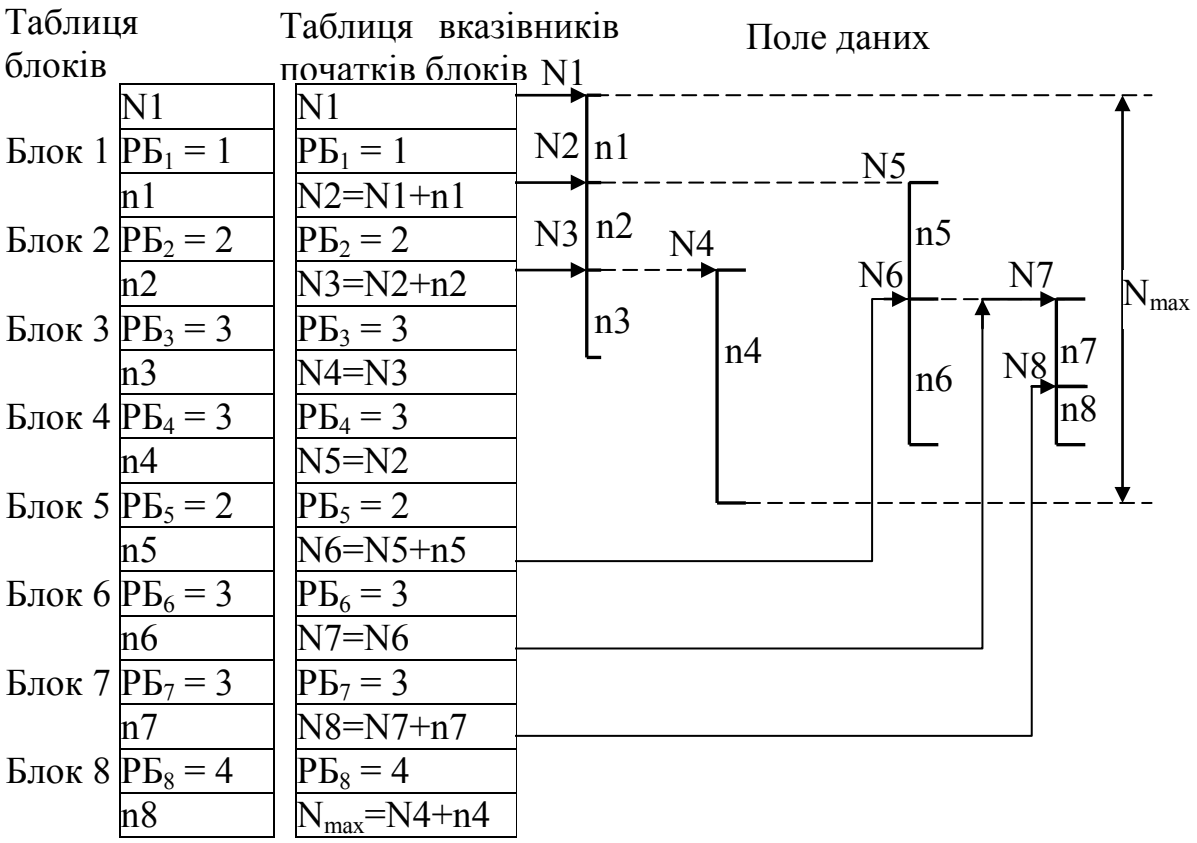


Рисунок 2.22 – Приклад статичного розподілу пам’яті

Потім таблиця блоків перетвориться в таблицю вказівників на частини пам’яті виділені під блоки. Замість *ni* формується вказівник на початок секції

пам'яті для n-го блока. Для першого блока це $N1$, для другого, $N1 + n1 = \max(Ni + ni)$ максимальне значення – остання зайнята комірка.

Якщо в блоці описати також і масиви, то вони можуть бути розміщені в області, відведеній для блока, проте часто масиви розташовують окремо в полі масивів (іноді їх навіть виносять в зовнішню пам'ять). Вважатимемо, що масиви «винесені».

Другий етап – призначення адрес скалярним змінним всередині кожного блока. Для цього використовується таблиця ідентифікаторів.

У загальному випадку таблиця ідентифікаторів містить:

- ідентифікатор або посилання на таблицю імен;
- порядковий номер блока – i;
- рівень блока РБ$_i$;
- порядковий номер ідентифікатора даного класу в блоці – j;
- ознаку, що визначає клас і тип ідентифікатора. (Класи: проста змінна, масив, мітка, процедура. Для простих змінних вказуються типи: дійсний, цілий, логічний і так далі)

Після складання таблиці вказівників на виділені під блоки частини пам’яті відносні адреси (порядкові номери ідентифікаторів) простих змінних j перераховуються в абсолютні адреси за формулою $\mathrm{ADDR} = Ni + j$. Абсолютні адреси заносяться в таблицю ідентифікаторів замість відносних. При статичному розподілі пам'яті під масиви виконують аналогічний перерахунок: на першому етапі кожному блоку виділяється секція пам'яті, на другому призначаються початкові адреси масивів всередині кожного блока.

Відмінність полягає в тому, що для масиву треба визначити не тільки його початкову адресу, але і місце зберігання вектора вказівників на рядки, що забезпечує обчислення адрес елементів масиву (обидва ці параметри заносяться в таблицю ідентифікаторів для кожного масиву).

Після розподілу пам'яті для скалярних змінних і визначальних векторів статичний розподіл пам'яті під масив можна провести таким чином:

- визначається максимальна кількість елементів кожного масиву;
- формується таблиця блоків (так само, як і для скалярних змінних);
- визначається вказівник початку поля масивів (якщо поле масивів слідує за полем опису скалярних змінних, то це $N_{\max}$);
- таблиця блоків перетворюється на таблицю вказівників на початки блоків (таку саму, як для скалярних змінних);
- для всіх блоків з номерами $i=1,\ldots,m$ визначається адреса початку кожного масиву за формулами:

$$b_1 = Ni + 1$$
$$b_j = b_{j-1} + n_{j-1}, \qquad \forall j = 2,\ldots,k_j$$

де b_j – адреса першого елемента масиву з номером j;

$j=2,\ldots,k_j$ (k_j – кількість масивів в i-ому блоці);

Ni – вказівник на початок секції пам'яті, що виділена для i-ого блока;

n_j – максимальна кількість елементів j-ого масиву.

Недоліком статичного розподілу є неекономний розподіл пам'яті: під масив завжди виділяється пам’ять, що необхідна для розміщення заданої максимальної кількості елементів масиву, при цьому масив не завжди заповнюється повністю. Перевагою такого розподілу є економія часу при виконанні програми. Однак, якщо для масивів використовується зовнішня пам'ять, то перевага зводиться нанівець через додаткові витрати, тому найчастіше в сучасних мовах програмування використовується динамічний розподіл пам'яті [26].

2.2.3. Динамічний розподіл пам'яті

Динамічний розподіл дозволяє мати єдину суцільну область пам'яті для всіх змінних, масивів, процедур і забезпечує простий спосіб обробки рекурсивних процедур. Крім того, наявність індексних регістрів у сучасних комп’ютерах

істотно полегшує динамічний розподіл пам'яті при виконанні програми. Це робить динамічний розподіл більш популярним.

Радикальним засобом реалізації динамічного розподілу є стек [27]. Розглянемо такий розподіл на прикладі.

Приклад 2.8

Нехай програма має наступну структуру (Рисунок 2.23).

```
p:  begin real a,b;
      real procedure P1(x,y);
        real x,y;
        begin real u,v;
        end;
      end p1;
      a:=p1(a,b);
  m1: begin real c,d;
    m2: begin real f..;
        d:=p1(c,f);
    m3: end m2;
    n2:
  end m1;
end.
```

Рисунок 2.23 – Приклад 2.8 програма

Подамо структуру програми у вигляді дерева (Рисунок 2.24).

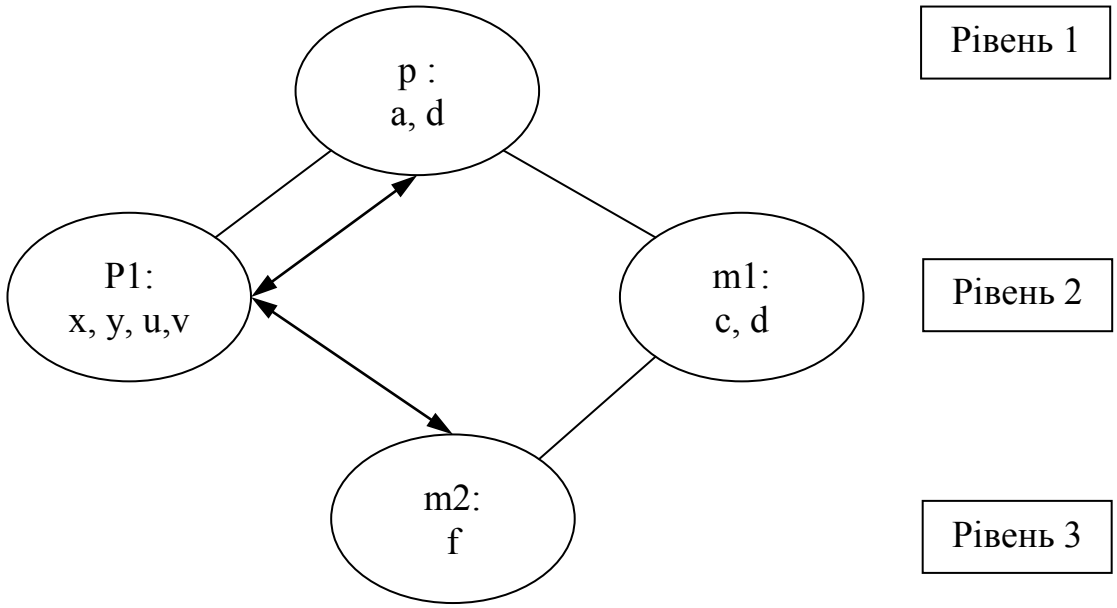


Рисунок 2.24 – Приклад 2.8 структури програми

Будемо вважати, що підпрограма помічена міткою P1, а основна програма має мітку p. Звернення до процедури P1 відбувається з зовнішнього блока p (дія a:=p1(a,b)) та з блока m2 (дія d:=p1(c,f)). Перерозподіл пам'яті відбувається при вході в блок відповідної процедури або функції. Вказане стрілками (Рисунок 2.24) переміщення по дереву пов'язане з динамічним перерозподілом пам'яті.

Розглянемо спосіб реалізації динамічного розподілу для даного прикладу. Всі дані, що використовуються в програмі, розміщуються в стеку. До входу в зовнішній блок стек вільний, вказівник стека (ВС) містить адресу першої вільної комірки (Рисунок 2.25).

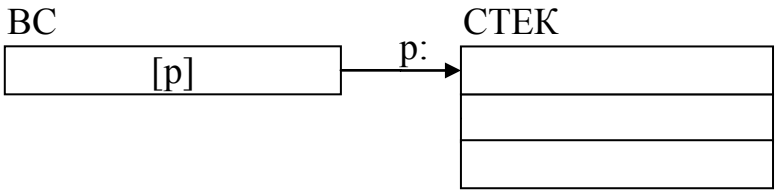


Рисунок 2.25 – Початковий стан стека

У початковому стані в ВС записано посилання на початок стека, відповідне мітці p, тобто початку програми.

Дії з підготовки стека відповідають підпрограмі ПОЧАТКОВИЙ ВХІД (Рисунок 2.26).

```
Стек(1) := ∅;
i:=1;
ВПБ(1) := ВС
ВС := ВС + n_p + 1
```

Рисунок 2.26 – ПОЧАТКОВИЙ ВХІД

При вході в блок p у вершині стека виділяється комірка для розміщення ознаки початку стека (∅), а потім виділяються комірки для всіх описаних і робочих змінних. Кількість комірок n_p. Також використовується комірка для зберігання змінної i – значення рівня блока (РБ) і масив вказівників на початки блоків (ВПБ) (Рисунок 2.27). Для підвищення швидкодії для елементів масиву ВПБ використовуються індексні регістри.

Адреси змінних, що входять в зовнішній блок, обчислюються як ВПБ(1) + j, де j – відносна адреса змінної (її номер всередині блока), визначена при трансляції.

У зовнішньому блоці виконується присвоєння a := P1(a,b); тобто звернення до процедури (функції), що відповідає переходу по дереву до блока P1 і перерозподілу пам'яті.

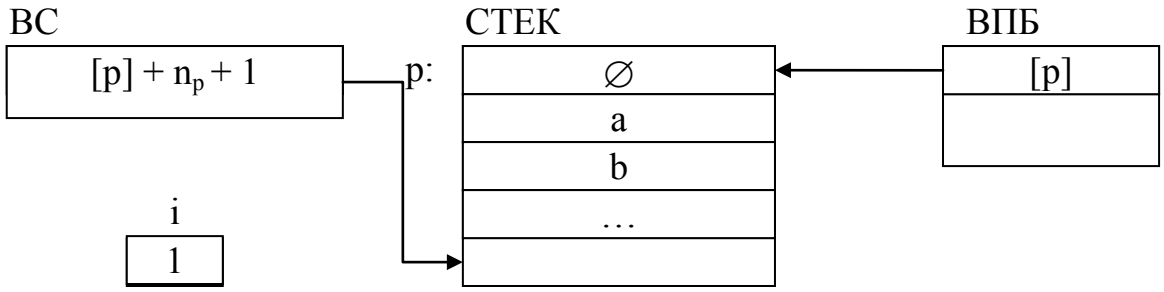


Рисунок 2.27 – Стан після входу в блок p

При вході в блок P1, що відповідає процедурі (функції), в (ВС+1)-ій комірці стеку формується запис, що складається з трьох компонентів:

$$ВПБ(РБ_k\text{-}1),\ ВПБ(i),\ i,$$

де $ВПБ(РБ_k\text{-}1)$ – вказівник на початок **цільового блока**, в якому знаходиться описана процедура (або, в загальному випадку, той блок, на який виконується перехід), k – номер цього блока, $РБ_k$ – його рівень;

ВПБ(i) – вказівник на початок **вихідного блока**, з якого відбулося звернення до процедури, тобто того блока, з якого в даний момент відбувається вихід;

i – рівень блока, з якого відбулося звернення до процедури.

Це сполучна інформація, що пов'язує блоки між собою і забезпечує повернення після виконання процедури (функції).

В даному випадку перші два параметри мають значення p, тобто формується запис:

[p], [p], 1.

Потім i набуває значення рівня блока, в який відбувся вхід:

$$i := РБ_{p1} = 2.$$

Формується новий елемент масиву ВПБ – ВПБ(i) (тобто в даному випадку ВПБ(2)), в який заноситься посилання на сполучну інформацію: на комірку з міткою P1. Вказівник стека збільшується з урахуванням розміщення змінних з процедури (Рисунок 2.28). Ці дії можна представити як виконання двох підпрограм: ПФ – процедура-функція та ВХІД – вхід в блок (Рисунок 2.29).

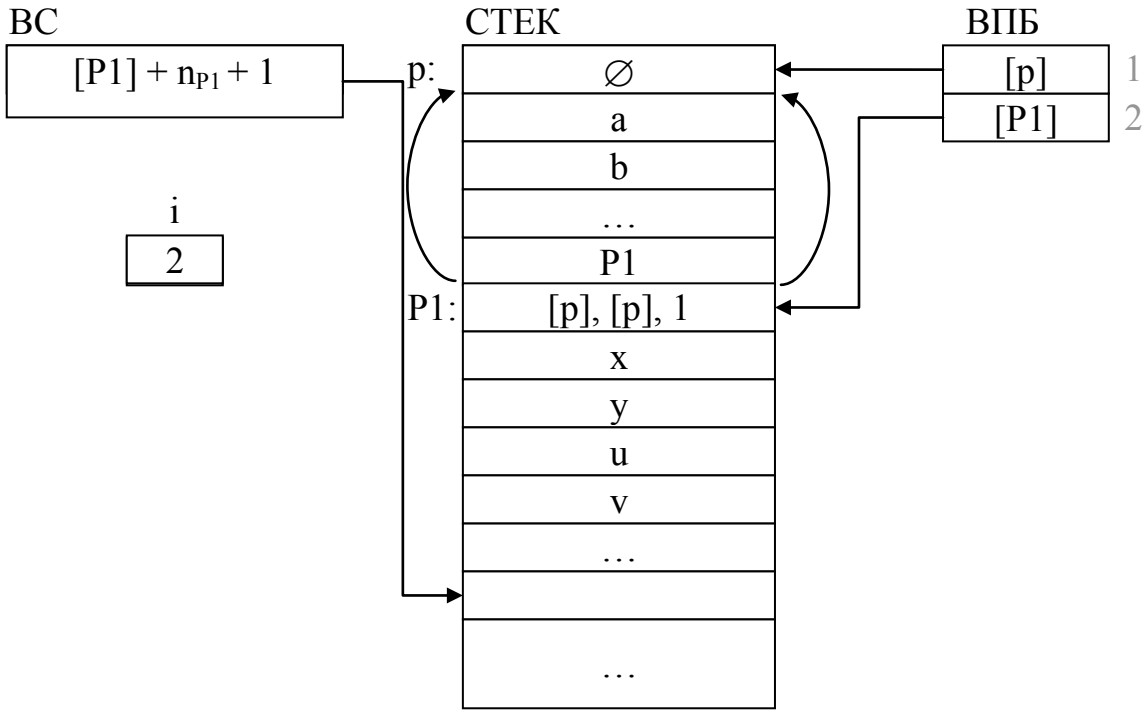


Рисунок 2.28 – Стан після звернення до процедури (функції) P1 з зовнішнього блока

```
ПФ:   ВС := ВС+1
ВХІД: стек(ВС) := (ВПБ(РБk-1), ВПБ(k), i);
      i := РБk // k – номер вхідного блока
      ВПБ(i) := ВС
      ВС := ВПБ(i)+nk+1 //ВС := ВС+nk+1
```

Рисунок 2.29 – Підпрограми ПФ та ВХІД

Звернення до змінних в тілі зовнішнього блока забезпечується адресами вигляду: ВПБ(1) + j, а до змінних в тілі процедури P1 – ВПБ(2) + j, де j – відно-

сна адреса змінної у відповідному блоці. Індекс масиву ВПБ дорівнює рівню відповідного блока. Так, якщо змінна y є другою в блоці P1, її адреса буде такою: ВПБ(2) + 1 = [P1] + 1.

Після виконання процедури (функції), визначення результуючого значення та виходу з неї змінні, локалізовані в цільовому блоці P1, стають недоступними. Натомість відновлюється доступ до змінних, що розміщені в блоці, з якого відбулося звернення (вихідного блоку). Для реалізації зміни доступу необхідно відновити стан стека, яким він був до виклику до процедури. Для цього використовуються сполучні дані:

$$(a_1^s, a_2^s, s),$$

де a_1^s – вказівник на початок цільового блока (в якому описана процедура);

a_2^s – вказівник на початок вихідного блока (з якого була викликана процедура);

s – рівень вихідного блока.

Перш за все слід змінити значення відповідної комірки масиву вказівників на початки блоків:

$$\text{ВПБ}(s) := a_2^s,$$

це робить доступними змінні блока рівня s ,при цьому, якщо $s > 2$, необхідно зробити доступними всі змінні, що є глобальними по відношенню до блока рівня s. Для цього будується ланцюжок перших посилань:

$$a_1^s \rightarrow a_1^{s-1} \rightarrow \ldots \rightarrow a_1^1.$$

Після всі елементи цього ланцюжка, окрім останнього, по черзі записуються в масив ВПБ. Ці дії виконує підпрограма виходу з процедури ВИХІД (Рисунок 2.30).

В даному випадку сполучна інформація – це [p], [p], 1. Тому відновлюється стан (Рисунок 2.27). Після входу в блок m1 стек змінюється відповідно до

процедури ВХІД (Рисунок 2.29) і система переходить в наступний стан (Рисунок 2.31).

```
ВИХІД:  ВС := ВПБ(i)-1
        Зі СТЕК(ВПБ(i)) беремо сполучні дані (a1s, a2s, s)
        if i = 1 then goto M2 else
          ВПБ(s) := a2s;
          i := s;
          j := s;
          M1: if j=2 then goto M2;
              Зі СТЕК(ВПБ(j)) взяти (a1s, a2s, s)
              j := s;
              ВПБ(i) := a1s;
              goto M1;
M2:
```

Рисунок 2.30 – Обробка виходу з процедури

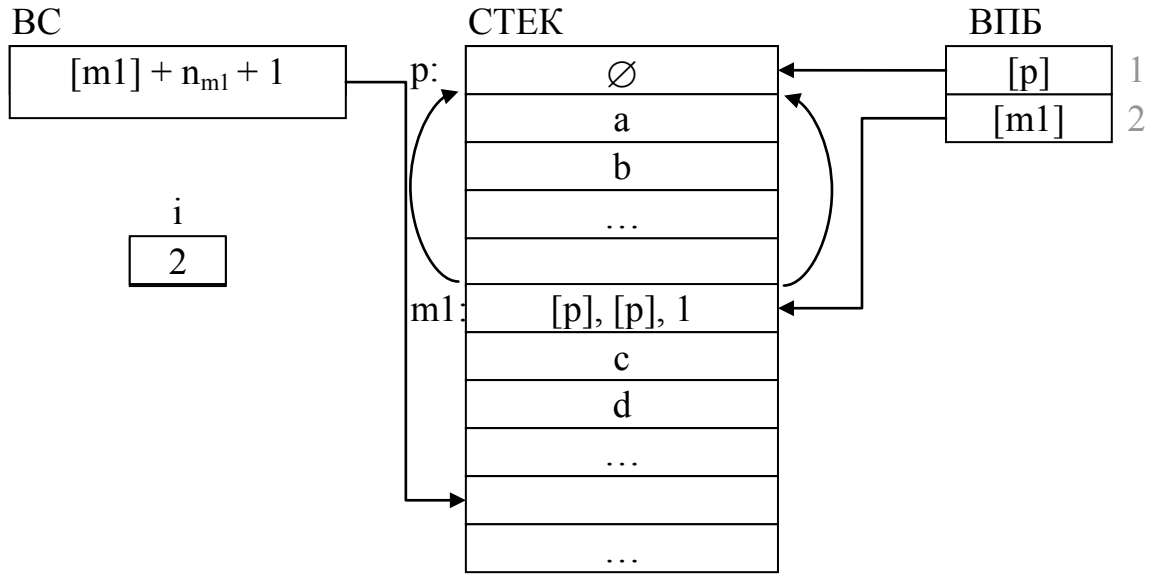


Рисунок 2.31 – Стан після входу в блок m1 з зовнішнього блока

При вході в блок m2, знову спрацьовує процедура ВХІД і система набуває іншого вигляду (Рисунок 2.32).

В блоці m2 наявне звернення до процедури P1: d ≔ P1(c, f);. Для обробки цього звернення стек змінюється відповідно до підпрограми ПФ та ВХІД (Рисунок 2.29), після чого він набуде такого вигляду (Рисунок 2.33).

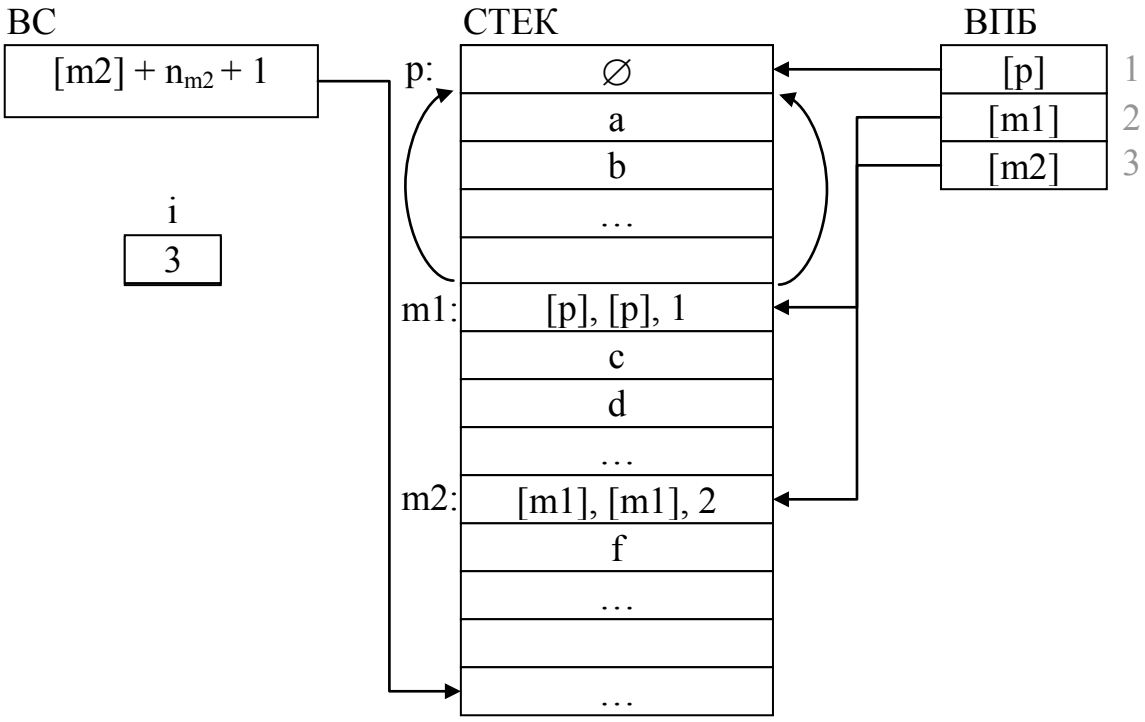


Рисунок 2.32 – Стан після входу в блок m2

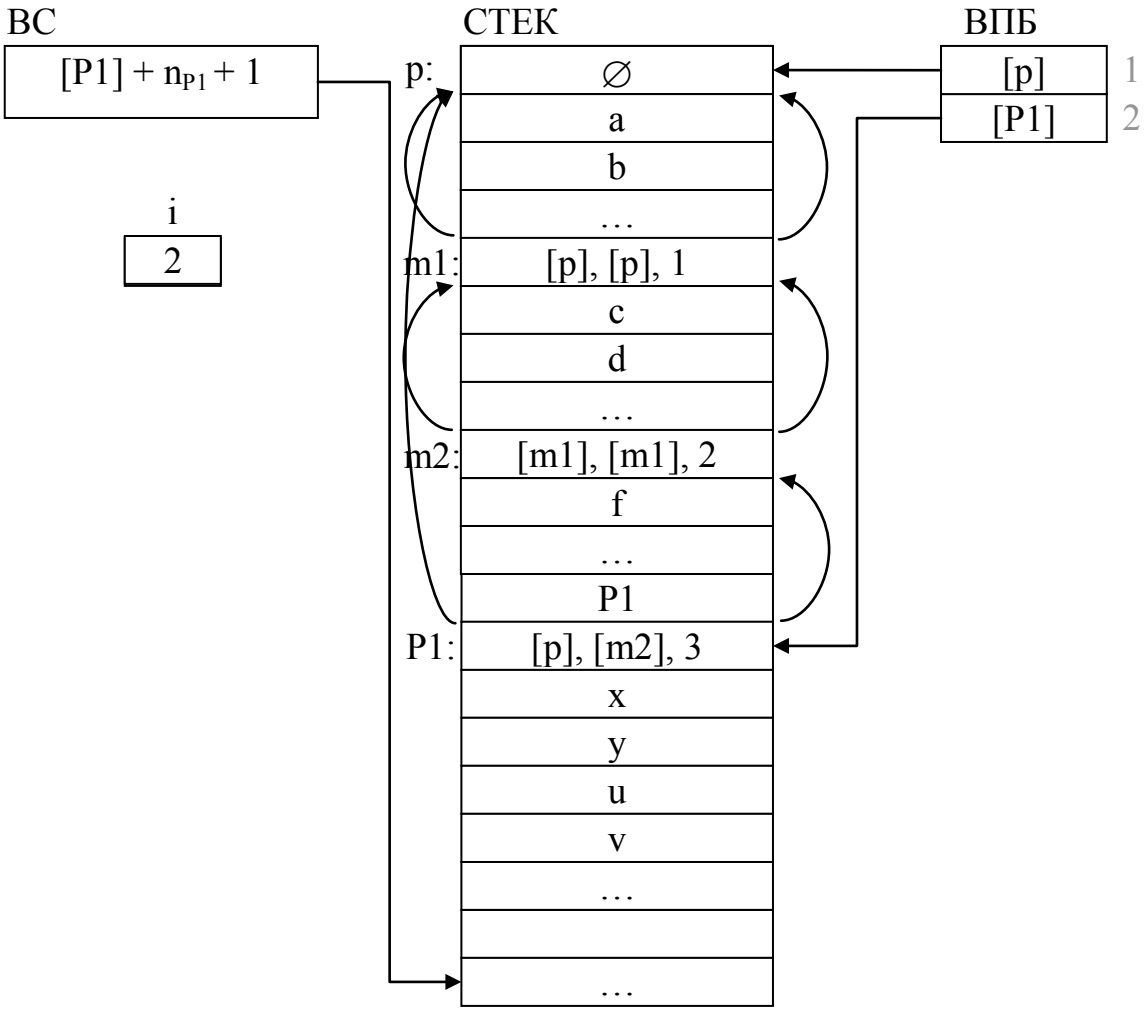


Рисунок 2.33 – Стан після входу в процедуру P1 з блока m2

Під час виконання операторів в тілі процедури-функції P1 усі змінні, що локалізовані в блоках m1 та m2, стають недоступними, масив ВПБ уже не містить вказівників на початки цих блоків.

В той самий час ланцюжок других посилань в сполучних даних пов'язує всі блоки, які почали виконуватися, але ще не завершилися. Цей ланцюжок називають **динамічним**.

Ланцюжок перших посилань зі сполучних даних пов'язує усі доступні в даний момент змінні. Він не залежить від ходу виконання програми і визначається виключно структурою програми, тому цей ланцюжок називають **статичним**.

> Статичний та динамічний ланцюжки співпадають, якщо нема звернень до процедур чи функцій або, якщо процедура-функція описана в тому самому блоці, з якого вона була викликана.

Після виконання процедури-функції P1 в результаті обробки виходу з процедури (Рисунок 2.30) стек повинен прийняти вигляд, що відповідає попередньому стану (Рисунок 2.32). Таким чином, доступ до змінних, описаних в блоках m1 та m2 відновиться. Перевіримо, чи справді це станеться.

Перед виконанням підпрограми виходу з процедури-функції (Рисунок 2.30) система перебуває в наступному стані (Рисунок 2.33). В ході процедури ВИХІД спочатку змінюється вказівник стеку відповідно до i-го значення масиву ВПБ, оскільки в даний момент i = 2:

$$\text{ВС} := \text{ВПБ(i)-1} = \text{ВПБ(2)-1} = [\text{P1}]\text{-1}.$$

Далі зі стека за адресою ВПБ(i) (тобто в даному випадку ВПБ(2)) беремо сполучні дані (a1s, a2s, s) = ([p], [m2], 3). Оскільки $i \neq 1$, в s-ту комірку масиву ВПБ заносимо a2s, що в поточний момент дорівнює [m2]. Далі змінним i та j присвоюється значення s, тобто 3. Оскільки $j \neq 2$, то за адресою ВПБ(j) зчитуються сполучні дані (a1s, a2s, s), в даний момент ВПБ(j) = ВПБ(3) = = [m2], а в стеку за адресою [m2] знаходяться дані ([m1], [m1], 2), отже тепер

a1s = [m1], a2s = [m1], s = 2. В змінну j заноситься s, тобто j = 2. В i-ий елемент масиву ВПБ заноситься поточне значення a1s = [m1]. Далі виконується повторна перевірка рівності j = 2, оскільки на цей раз, це правда, то процедура виходу завершується (перехід на мітку m2). Дійсно, ці дії привели систему до попереднього стану (Рисунок 2.32).

При переході до мітки n2 відбувається вихід з блока відповідно до підпрограми ВИХІД З БЛОКА (Рисунок 2.34).

```
ВИХІД З БЛОКА:  i := РБs // РБs – рівень цільового блока
                ВС ≔ ВПБ [i +1]
```

Рисунок 2.34 – Обробка виходу з блока

При виході через процедуру ВИХІД З БЛОКА рівень цільового блока на одиницю менший за рівень вихідного блока.

При виході по мітці інформація про рівень цільового блока повинна міститися в мітці. Вихід з блока по мітці має супроводжуватися виконанням підпрограми ВИХІД З БЛОКА.

При динамічному розподілі пам’яті рекурсивні процедури чи функції обробляються так само, як усі інші. Тому, якщо описана раніше процедура-функція P1 звернеться сама до себе, стек набуде наступного вигляду (Рисунок 2.35). Початок рекурсивного звернення в стеку позначено міткою P1′.

Для практичної реалізації динамічного розподілу пам’яті необхідно в ході трансляції включити в цільовий код команди, що забезпечують виконання наведених підпрограм:

- ПОЧАТКОВИЙ ВХІД (Рисунок 2.26) – при вході в зовнішній блок;
- ВХІД (Рисунок 2.29) – при вході в кожний блок;
- ПФ та ВХІД (Рисунок 2.29) – при зверненні до процедури-функції;
- ВИХІД (Рисунок 2.30) – при виході з процедури-функції;
- ВИХІД З БЛОКА (Рисунок 2.34) – при виході з блока через його кінець або по мітці.

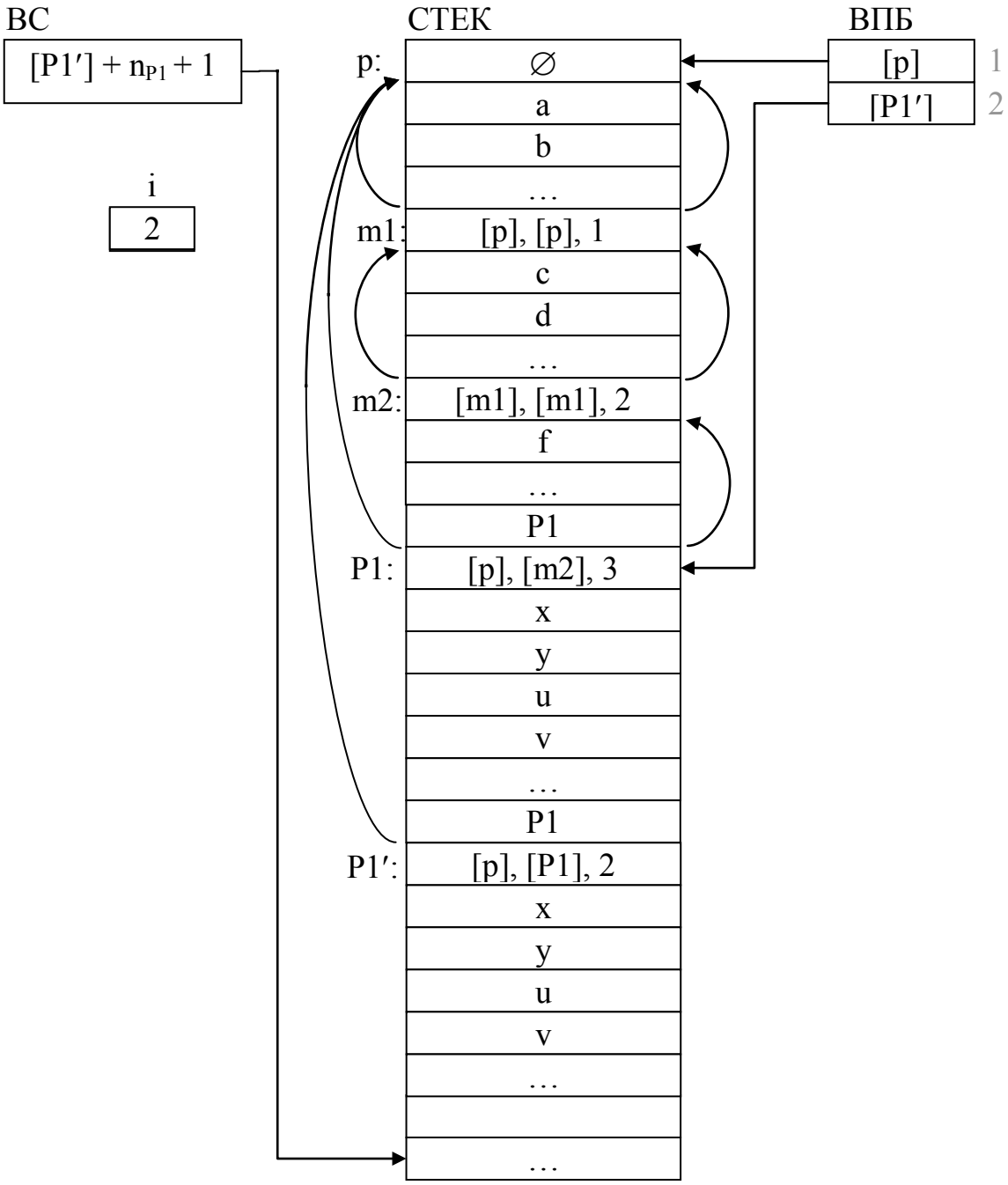


Рисунок 2.35 – Стан після входу в процедуру P1 з процедури P1

Програмування цих підпрограм, а також звернення до них забезпечується інформацією, що міститься в таблиці блоків. На відміну від статичного розподілу пам’яті, записи, що вказують кількість змінних в блоці, обчислюється наново кожного разу при вході в блок. Для цього в цільову програму слід включити команди, що дозволяють обчислити кількість змінних n_j перед кожним виконанням підпрограми ВХІД.

2.2.4. Переклад в ПОЛІЗ блоків та оголошень типу

Результати обробки оголошень змінних частково фіксуються в таблицях компілятора, частково – в цільовій програмі. В різних компіляторах оголошення обробляють на різних етапах трансляції. Однак в будь-якому разі обробка оголошень повинна передувати генерації машинних команд (речень цільової мови), що відповідають операторній частині програми (або блока). Тому більшість компіляторів виконують обробку оголошень одразу після перекладу програми на внутрішню мову подання.

Формат запису таблиці ідентифікаторів може бути наступним (Рисунок 2.36).

Ідентифікатор
Номер блока – і
Рівень блока – РБ
Відносна адреса (порядковий номер) ідентифікатора в блоці
Ознака (тип або клас змінної)

Рисунок 2.36 – Заголовки таблиці ідентифікаторів

При динамічному розподілі пам’яті в задачу обробки блоків окрім визначення номерів та рівнів блоків для таблиці ідентифікаторів входить формування команд, що забезпечать виконання підпрограм обробки входу та виходу до блоків та процедур (Рисунок 2.26, Рисунок 2.29, Рисунок 2.30, Рисунок 2.34).

Для отримання проміжного подання програми з урахуванням обробки блоків у ПОЛІЗ слід ввести додаткові операції:

- операція **ТИП**;
- операція початку блока **ПБ**;
- операція завершення оголошень **ЗО**;
- операція завершення блока **ЗБ**.

Операція **ТИП** має один аргумент і записується так:

k ТИП

де **k** – кількість операндів простих змінних у списку вказаного типу.

Наприклад, запис **3 integer** означає, що в списку типу **integer** описано три змінні.

Операції **ПБ** та **ЗО** мають по два операнди: номер блока (**i**) та рівень блока (**РБ**). Операція **ЗБ** не має операндів.

З використанням операції **ТИП** оголошення змінних **real a, b, c** транслюється в ПОЛІЗ як **a b c 3 real**.

Межі блоків також перетворюються на відповідні команди.

Приклад 2.9

Нехай на вхід компілятора надійшла програма з наступною структурою (Рисунок 2.37). Тоді отримаємо ПОЛІЗ наступного вигляду (Рисунок 2.38).

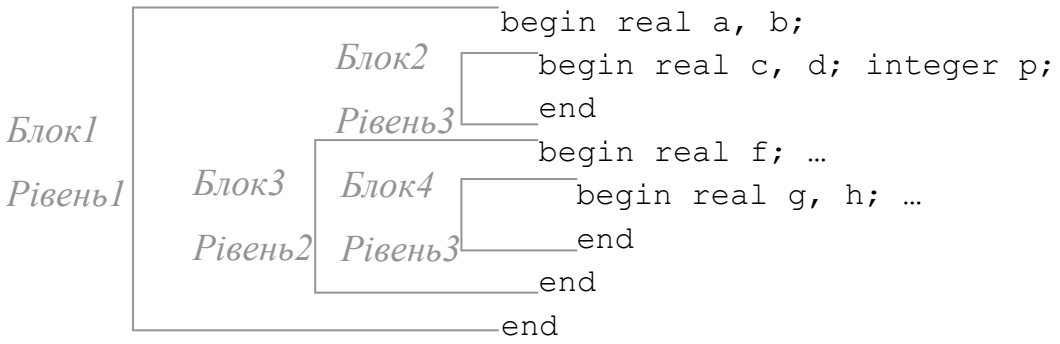


Рисунок 2.37 – Приклад програми з блоками та оголошеннями

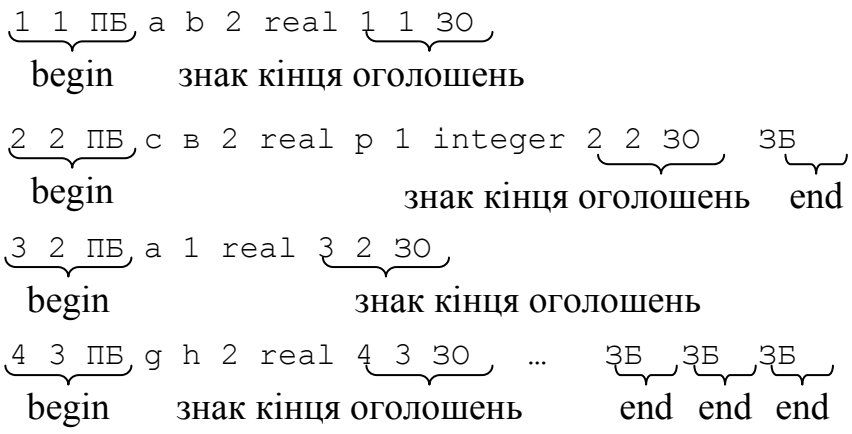


Рисунок 2.38 – Приклад ПОЛІЗ блоків та оголошень

Фрагмент **i РБ ПБ** (в прикладі це частини: **1 1 ПБ**, **2 2 ПБ**, **3 2 ПБ** та **4 3 ПБ**) генерується по службовому слову **begin**. Під час генерації цільової програ-

ми операція початку блока **ПБ** не породжує команд в цільовій програмі. Відповідно до цієї операції виконують наступні дії:

- номер блока зберігається в змінну **i**;
- значення рівня блока – в змінну **РБ**;
- поточний номер змінної в блоці встановлюється рівним нулю (**j ≔ 0**).

Значення цих змінних використовується операціями **ТИП** при формуванні записів таблиці ідентифікаторів.

Операція **ТИП** також не породжує команд в цільовій програмі. В ході трансляції ця операція, використовуючи значення змінних **i**, **РБ** та **j**, формує для кожного ідентифікатора, що є її операндом, запис в таблиці ідентифікаторів в наведеному вище (Рисунок 2.36) форматі. Перед занесенням в таблицю чергового ідентифікатора значення **j** збільшується на одиницю.

Операція **ЗО** формує в об'єктній програмі підпрограму ПОЧАТКОВИЙ ВХІД, якщо **i = 1**, або підпрограму ВХІД, якщо **i > 1**. Операція **ЗБ** формує підпрограму ВИХІД З БЛОКА. При перекладі вхідного рядка в ПОЛІЗ окрім змінних **i**, **РБ** та **j** використовується лічильник блоків та, можливо, інші змінні, що дозволяють визначити місце початку та завершення оголошень і блоків.

Якщо в мові використовуються динамічні типи даних, то останнім часом в обов'язки компілятора включають також автоматичне видалення динамічних змінних, на які уже не посилаються вказівники. Кажуть, що цим займається «збирач сміття». Очевидно, доречним є виклик «збирача сміття» після виходу з блока. З іншого боку, автоматичне прибирання зайвих динамічних даних може виконуватися в час найменшого завантаження машини. Через те, що дані видаляються автоматично, програміст-користувач не може передбачити час видалення даних, тому невдалою практикою є запис та зчитування адрес динамічних змінних з зовнішніх для програми джерел.

2.3. Завдання для самоконтролю

Завдання 2.1. Дана упорядкована таблиця, в якій заповнено 9 позицій (Таблиця 2.9).

Таблиця 2.9

1	2	3	4	5	6	7	8	9	10
ab	ba	cad	del	did	e	f1	p	qun	

Використовуючи метод бінарного пошуку, визначити за скільки порівнянь

а) буде знайдено ідентифікатор **f1**;

б) буде видане повідомлення про відсутність в таблиці ідентифікатора **k**;

в) буде знайдено ідентифікатор **cad**;

г) буде знайдено ідентифікатор **p**;

ґ) буде знайдено ідентифікатор **qun**;

д) буде видане повідомлення про відсутність в таблиці ідентифікатора **cp**;

Завдання 2.2. Методом хешування в таблицю занесено символи **S1**, **S2**, **S3**, **S4**, **S5** за адресами: 2, 3, 5, 7, 8. Визначити за якою адресою буде занесено символ **S6** та скільки пробоїв таблиці для цього знадобиться, якщо

а) **h0(S6) = 2**, і конфлікти розв'язуються лінійним рехешуванням;

б) якщо **h0(S6) = 3**, і конфлікти розв'язуються методом квадратичного рехешування, при чому значення коефіцієнтів наступні: **a=1, b=1, c=0**;

в) **h0(S6) = 7**, і конфлікти розв'язуються лінійним рехешуванням;

г) **h0(S6) = 3**, і конфлікти розв'язуються лінійним рехешуванням;

ґ) **h0(S6) = 5**, і конфлікти розв'язуються лінійним рехешуванням;

д) **h0(S6) = 5**, і конфлікти розв’язуються методом квадратичного рехешування, при чому значення коефіцієнтів наступні: **a=1, b=1, c=0**;

е) **h0(S6) = 7**, і конфлікти розв’язуються методом квадратичного рехешування, при чому значення коефіцієнтів наступні: **a=1, b=1, c=0**;

є) **h0(S6) = 8**, і конфлікти розв’язуються методом квадратичного рехешування, при цьому значення коефіцієнтів наступні: **a=1, b=1, c=0**.

Завдання 2.3. Методом ланцюжків в таблицю занесено символи **S1**, **S2**, **S3**, **S4**, для яких хеш-функція визначила адреси 4, 1, 3, 6, відповідно. Тобто таблиці мають вигляд (Рисунок 2.8). Нехай на вхід послідовно надходять два символи: **S5**, **S6**. Визначити вигляд таблиць після запису цих двох символів за умови, що

а) для **S5** хеш-функція формує адресу 2, для **S6** – 1;

б) для **S5** хеш-функція формує адресу 2, для **S6** – 3;

в) для **S5** хеш-функція формує адресу 2, для **S6** – 6;

г) для **S5** хеш-функція формує адресу 5, для **S6** – 1;

ґ) для **S5** хеш-функція формує адресу 5, для **S6** – 3;

д) для **S5** хеш-функція формує адресу 5, для **S6** – 4;

е) для **S5** хеш-функція формує адресу 5, для **S6** – 6;

є) для **S5** хеш-функція формує адресу 2, для **S6** – 4;

ж) для **S5** хеш-функція формує адресу 5, для **S6** – 5;

з) для S5 хеш-функція формує адресу 2, для **S6** – 2;

Завдання 2.4. Як будуть розміщуватися масиви, перелічених нижче розмірностей, відповідно лінійного відображення?

а) 3×8; б) 7×4; в) $3\times 8\times 1$; г) $3\times 8\times 2$;

ґ) $[0..1,2..5]$; д) $[-3..3,-1..1]$; е) $[0..4,-2..1,1..5]$

Завдання 2.5. Реалізувати масиви з завдання 2.4, використовуючи вказівники на рядки.

Завдання 2.6. Реалізувати масиви з завдання 2.4, використовуючи вектори Айліффа.

Завдання 2.7. Надати схему розміщення в пам’яті даних наступних програм (Рисунок 2.39) при статичному розподіленні пам’яті.

```
begin n1
  begin n2
    begin n3
    end
    begin n4
    end
  end
  begin n5
    begin n6
    end
  end
end
```

а)

```
begin n1
  begin n2
    begin n3
    end
    begin n4
    end
    begin n5
      begin n6
      end
    end
  end
end
```

б)

```
begin n1
  begin n2
    begin n3
    end
  end
  begin n4
    begin n5
    end
    begin n6
    end
  end
end
```

г)

```
begin n1
  begin n2
    begin n3
    end
    begin n4
      begin n5
      end
      begin n6
      end
    end
  end
end
```

ґ)

```
begin n1
  begin n2
    begin n3
    end
    begin n4
    end
  end
  begin n5
    begin n6
    end
  end
end
```

в)

```
begin n1
      begin n2
        begin n3
          begin n5
          end
          begin n6
          end
        end
        begin n4
        end
      end
    end
```

д)

```
begin n1
  begin n2
    begin n3
    end
    begin n4
    end
  end
  begin n5
    begin n6
      begin n7
      end
    end
    begin n8
    end
end
```

е)

```
begin n1
  begin n2
    begin n3
      begin n4
      end
      begin n5
      end
    end
  end
end
```

є)

```
begin n1
  begin n2
    begin n3
    end
  end

  begin n4
    begin n5
    end
  end
end
```

ж)

Рисунок 2.39

Завдання 2.8. При динамічному розподілі пам’яті початковий стан системи для програми (Рисунок 2.40) відповідає розглянутому раніше прикладу (Рисунок 2.25). Після входу в зовнішній блок система набуде вигляду (Рисунок 2.27). Визначити якого вигляду набуде система після

а) звернення до функції P1 з зовнішнього блока;

б) виходу з функції P1 і повернення до зовнішнього блока (використовуючи алгоритм виходу з процедури (Рисунок 2.30));

в) входу в блок m1;

г) входу в блок m2;

ґ) переходу до мітки m3;

д) виклику функції P1 з блока m1;

е) виходу з функції P1 і повернення до блока m1;

є) переходу до мітки n2;

ж) виходу з зовнішнього блока.

Завдання 2.9. Зобразити структуру програми (Рисунок 2.40) у вигляді дерева.

```
p:  begin real a,b;
      real procedure P1(x,y);
        real x,y;
        begin real u,v;
        end;
      end P1;
      a:=P1(a,b);
  m1: begin real c,d;
    m2: begin real f..;
    m3: end m2;
    d:=P1(c,f);
    n2: end m1;
end.
```

Рисунок 2.40

ПРЕДМЕТНИЙ ПОКАЖЧИК

БІБЛІОГРАФІЯ

1. Медведєва, В. М. Транслятори: лексичний та синтаксичний аналізатори / В. М. Медведєва, В. А. Третяк. – К. : НТУУ "КПІ", 2012. – 148 с.

2. Рихтер, Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 2.0 на языке C#. Мастер-класс. [Пер. с англ.] / Дж. Рихтер – М. : Издательство «Русская Редакция» ; СПб. : Питер , 2007. – 656 с.

3. Diehl, S. A Formal Introduction to the Compilation of Java / Stephan Diehl // Software - Practice and Experience. – March, 1998. – Vol. 28(3). – P. 297-327.

4. Практикум «Оптимизирующие компиляторы» (на примере GCC) [Электронный ресурс]. – НГУ им. Лобачевского. – Режим доступа: compiler.itlab.unn.ru/uploads/compiler_practice.update19.doc.

5. Ахо, А.В. Компиляторы: Принципы, технологии, инструментарий / А.В.Ахо, М.С. Лам, Р. Сети [и др.].– 2-е изд. : пер с анлг. – М.[и др.]: ИД Вильямс, 2010.– 1184 с.

6. Маккиман, У. Генератор компиляторов / У.Маккиман, Дж. Хорнинг, Д. Уортман; пер. с англ. С. М. Круговой;[под.ред. и с предисл. В.М. Савинкова].– М. : Статистика, 1980.– 527 с.

7. Хантер, Р. Проектирование и конструирование компиляторов / Р. Хантер. – М. : Финансы и статистика, 1984.– 232 с. .

8. Пратт, Т. Языки программирования: разработка и реализация / Т. Пратт, М. Зелковиц.– СПб.: Питер, 2002.– 688 с.

9. Льюис, Р. Теоретические основы проектирования компиляторов / Р. Льюис, Д. Розенкранц, Р. Стирнз. – М. : Мир, 1976. – 655 с.

10. Теория и реализация языков программирования: Курс лекций / В.М. Курочкин, Л.Н. Столяров, Б.Г. Сушков, Ю.А. Флёров. – М. : МФТИ, 1973. – 144 с.

11. Мартыненко, Б.К. Языки и трансляции : Учеб. пособие / Б.К. Мартыненко. – Изд. 2-е, испр. и доп. – [СПб] : Изд-во С.-Петербургского университета, 2008. – 257 с.

12. Грис, Д. Конструирование компиляторов для цифровых вычислительных машин / Д. Грис. – М.: Мир, 1975.– 544 с.

13. Shaw, A. Lecture notes on a course in systems programming / A. Shaw. – Stanford : Computer Science Dept. Stanford Univ., 1966. – 209 p.

14. Dijkstra, E. W. Algol-60 Translation / E. W. Dijkstra. // Algol Bulletin. Amsterdam : Stichting Mathematisch Centrum , 1961, .– №10. – 11 p.

15. Muchnick, S. Advanced Compiler Design and Implementation / Steven Muchnick. – San Francisco [et al.] : Morgan Kaufman, 1997. – 856 p.

16. Теория и реализация языков программирования / В.А. Серебряков, М.П. Галочкин, Д.Р. Гончар, М.Г. Фуругян. – М. : МЗ Пресс, 2006. – 356 с.

17. Aho, A.U. Code generation using tree matching and dynamic programing / A.U. Aho, M. Ganapathi, S.W. Tjiang // ACM Trans. Progr. Languages and Systems.– 1989.– V.11. N 4.

18. Надежин, Д.Ю. Промежуточный язык Лидер (предварительное сообщение). Обработка символьной информации. / Д.Ю. Надежин, В.А. Серебряков, В.М. Ходукин. – М.: ВЦ АН СССР, 1987.– С. 50-63.

19. Mogensen, T. Basics of Compiler Design [Electronic resource] / Torben Mogensen. – Anniversary edition. – Copenhagen : Department of Computer Science. University of Copenhagen, 2010. – Access mode: http://www.diku.dk/~torbenm/Basics.

20. Bezdushny, A. The use of the parsing method for optimal code generation and common subexpression elimination / A. Bezdushny, V. Serebriakov // Techn. et Sci. Inform.– 1993.– V.12. N.1.– P.69-92.

21. Emmelman, H. BEG -a generator for efficient back-ends / H. Emmelman, R. Landweher, F.W. Schroer // ACM SIGPLAN.– 1989.– V. 11. N 4.– P. 227-237.

22. A complete, flexible compiler construction system / R.W. Gray, V.P. Heuring, S.P. Levi [et.al.] // CACM. – 1992. – V. 35, №2.– P. 193-208. .

23. Johnson, S.C. Yacc-yet another compiler-compiler[Electronical resource].– N.J. : Murray Hill, 1975. – Access mode : http://miffy.tom-yam.or.jp/2238/ref/yacc.pdf.

24. GAG: A Practical Compiler Generator (Lecture Notes in Computer Science) / U. Kastens, B. Hutt, E. Zimmermann. – Berlin [et al.] : Springer, 1982. – 157 p.

25. Юров, В.И. Assembler практикум / В.И. Юров. – СПб : ООО "Питер-пресс", 2007. – 400 с.

26. Акиньшин, А. Внутреннее устройство массивов в .NET [Елктронний ресурс] / А. Акиньшин. – 11.10.2013. – Режим доступу : http://aakinshin.blogspot.com/2013/10/dotnet-arrays-internal-structure.html. – Загол. з екрану.

27. Лебедев, В.Н. Введение в системы программирования / В.Н. Лебедев. – М.: Статистика, 1975. – 312 с.

НАВЧАЛЬНЕ ВИДАННЯ

Медведєва Валентина Миколаївна

Третяк Валерія Анатоліївна

Транслятори: внутрішнє подання програм та інтерпретація

Підп. до друку 18.03.2015 р. Формат $60\times84\,{}^{1}\!/_{16}$. Папір офс. Гарнітура Times.

Ум. друк. арк. 8,37. Обл. вид. арк. 13,91.

Наклад 150 пр. Зам. №

Віддруковано видавничо-поліграфічним підприємством «Текст»

(Св. про реєстрацію №7760 від 02.11.98

Св. ВВР видавничої продукції ДК 720 від 17.12.2001)

м. Київ, вул. Нагірна, 22