О. О. Шумейко
В. М. Кнуренко

Visual Prolog

Опануй на прикладах

О. О. Шумейко
В. М. Кнуренко

VISUAL PROLOG.
ОПАНУЙ НА ПРИКЛАДАХ

Навчальний посібник

Рекомендовано Міністерством освіти і науки України

Дніпропетровськ
Видавець Біла К.О.
2014

УДК 004.896
ББК 32.973-018.1
Ш 96

Рекомендовано Міністерством освіти і науки України
як навчальний посібник для студентів вищих навчальних закладів
(лист № 1/11-17387 від 13.11.2013 р.)

Рецензенти:

Байбуз О.Г., д-р техн. наук, професор, завідувач кафедри математичного забезпечення ЕОМ Дніпропетровського національного університету ім. О. Гончара;

Приставка П.О., д-р техн. наук, професор, завідувач кафедри прикладної математики Київського національного авіаційного університету;

Самохвалов С.С., д-р техн. наук, професор, завідувач кафедри прикладної математики Дніпродзержинського державного технічного університету.

Шумейко О. О.

Ш 96 Visual Prolog. Опануй на прикладах : навч. посіб. / О. О. Шумейко, В.М. Кнуренко. – Дніпропетровськ : Біла К. О., 2014. – 404 с.

ISBN 978-617-645-008-5

У посібнику викладено матеріал дисципліни «Логічне програмування» відповідно до програми підготовки студентів вищих навчальних закладів за напрямом «Комп'ютерна інженерія».

Розглянуто парадигми та короткий опис історії мови логічного програмування. Викладено основи мови Visual Prolog, опис роботи у середовищі візуальної розробки Visual Prolog 7.4, основні механізми уніфікації і пошуку з поверненням та повторення і рекурсію, списки, внутрішні бази фактів, бази даних, арифметику та ін.

Розглянуто зразки програм мовою Visual Prolog, додано питання та тестові завдання для самоконтролю знань.

Буде корисним для студентів комп'ютерного факультету вищих навчальних закладів, аспірантів, науковців та працівників комп'ютерної промисловості.

УДК 004.896
ББК 32.973-018.1

ISBN 978-617-645-168-6

ЗМІСТ

ВСТУП

Комп’ютерам ще далеко до досягнення мети – стати рівним партнером людини в його інтелектуальній діяльності. Але з історичної точки зору використання логіки як відповідного ступіню на цьому довгому шляху є природним і плідним, оскільки саме логіка супроводжує процес мислення людини з моменту зародження інтелекту.

Логіка давно використовується і при проектуванні комп’ютерів, і при аналізі комп’ютерних програм. Проте безпосереднє використання логіки в якості мови програмування, що називають логічним програмуванням, виникло порівняно недавно.

Логічне програмування будується не за допомогою деякої послідовності абстракцій і перетворень, які відштовхується від машинної архітектури фон Неймана і властивого їй набору операцій, а на основі абстрактної моделі, яка ніяк не пов’язана з будь-яким типом машинної моделі. Логічне програмування базується на переконанні, що не людину слід навчати мисленню в термінах операцій, а комп’ютер повинен виконувати дії, властиві людині.

Логічне або реляційне програмування відкрило появу мови Prolog. Існує ряд інших мов, які не отримали такого широкого визнання в колі програмістів. Це такі як KL0 та ShapeUp.

KL0 (від англ. «kernel-language version 0» – ядро-мова версії 0) – мова, в основу якої покладено розширення мови логічного програмування Пролог. До найбільш істотних механізмів Пролога, що не підтримуються в KL0, відносяться: засоби управління базою даних; засоби управління таблицею імен.

ShapeUp – мова, базовою основою якої є Пролог, розширений засобами зіставлення рядків.

До мов логічного програмування відносяться також: Дейталог, LogLisp, Lisp та багато інших. Але серед логічних найбільш розвиненою і поширеною мовою є саме Prolog.

Prolog («PROgramming in LOGic» – програмування в термінах логіки, українською Пролог) – декларативна мова програмування загального

призначення, пов’язана зі штучним інтелектом та математичною лінгвістикою. Пролог був створений в 1972 р. з метою поєднання використання логіки з представленням знань. З тих пір у нього з’явився ряд діалектів, що розширюють основу мови різними можливостями.

ПАРАДИГМИ

Стандарт мови даний в ISO/IEC 13211-1 (1995 рік).

Розрізняють три діалекти мови програмування Prolog: Edinburgh Prolog, ISO Prolog і Strawberry Prolog. Серед них найбільш широкого поширення набув так званий единбурзький діалект (Edinburgh Prolog). Цей же діалект став основою стандарту ISO Prolog.

Пролог підтримує наступні парадигми:

- Парадигма обчислення з відкатами (мови з підтримкою цієї парадигми підтримують в явному вигляді відкат до попереднього стану);
- Декларативна парадигма програмування (визначає процес обчислень за допомогою опису логіки самого обчислення, а не логіки програми, що управляє. Декларативне програмування є протилежністю імперативного програмування: перше описує, що необхідно зробити, а друге – як саме це зробити);
- Логічна парадигма програмування (передбачає використання математичної логіки для розробки програм. Логічне програмування є окремим випадком декларативного програмування, оскільки програміст задає тільки набір формул, а ухвалення рішень про організацію обчислень приймається компілятором);
- Рефлексивна парадигма програмування (різновид метапрограмування, що передбачає написання програм, які можуть змінювати свою власну поведінку. Можливість обробки інструкцій так само, як і даних, є одним з ключових моментів архітектури фон Неймана. Різниця між ними виявляється тільки в

тому, як саме їх обробляє компілятор. У більшості мов інструкції виконуються, а дані обробляються. Рефлексивне програмування дозволяє обробку інструкцій перед їх виконанням. Таким чином, послідовність інструкцій, що підлягають виконанню, може бути сформована в процесі виконання на підставі вхідних даних і іншої інформації, доступної тільки на цьому етапі);

- Функціональна парадигма програмування (процес обчислення трактується як обчислення значень функцій в математичному розумінні, тобто тих, чий єдиний результат роботи полягає в значенні, що повертається. Спосіб рішення задачі описується за допомогою залежності функцій однієї від іншої, у тому числі і рекурсивно, але без вказівки послідовності кроків).

Головною парадигмою, реалізованою в мові Пролог, є логічне програмування, в якому програма є множиною пар (логічна умова, нові факти). Програміст залишається не обізнаним про методи, що використовуються при обчисленні, і послідовності виконання елементарних дій. Велика доля відповідальності за ефективність обчислень в логічному програмуванні перекладається на транслятор мови програмування. При використанні логічного програмування в програмах описується спосіб рішення поставленої задачі, а не вказуються кроки для отримання результату.

Подальші реалізації мови Visual Prolog додають в мову нові парадигми, наприклад, об’єктно-орієнтоване або кероване подіями програмування, іноді навіть з елементами імперативного стилю.

В іншому Пролог не відрізняється від традиційних мов програмування. Як і у випадку програми, написаної на будь-якій іншій мові, Пролог-програма призначена для вирішення окремого завдання.

Мова Пролог заснована на логіці диз’юнктів Хорна, що є підмножиною логіки предикатів першого порядку, та сприймає в якості програми деякий опис завдання і автономно проводить пошук рішення, користуючись механізмом

бектрекінгу (автоматичного повернення при пошуку рішення) і уніфікацією (зв'язування визначеного виклику з конкретною фразою).

Пролог відноситься до так званих декларативних мов, що вимагають від автора уміння скласти формальний опис ситуації. Тому програма на Пролозі не є такою в традиційному розумінні, оскільки не містить управляючих конструкцій, таких як if … then, while ... do. У Пролозі також відсутній оператор привласнення, у даній мові програмування задіяні інші механізми. Завдання описується в термінах фактів і правил, а пошук рішення Пролог бере на себе за допомогою вбудованого механізму логічного висновку.

Розробка програми починається частіше в середовищі інтерпретатора, оскільки він дозволяє виконувати, відлагоджувати і безперервно змінювати текст програми. Після того, як розробка окремих фрагментів програми закінчена, вони можуть бути відкомпільованими і вбудованими потім в інтерпретатор. Таким чином, подальша розробка програми може продовжуватися вже з використанням відкомпільованих сегментів програми.

Перелік можливих синтаксичних конструкцій Прологу невеликий і, в цьому сенсі, мова проста для вивчення.

Пролог реалізований практично для всіх відомих операційних систем і платформ. До числа операційних систем входять OS для мейнфреймів, все сімейство Unix, Windows, OS для мобільних платформ. Багато сучасних реалізацій мови мають внутрішнє розширення за рахунок ООП-архітектури. Окрім пропрієтарних (обмежених на використання та поширення) рішень, існують вільні реалізації Пролог.

ІСТОРІЯ

Початок історії мови Пролог відноситься до 70-х років XX століття. Першими дослідниками, які зайнялися розробкою ідеї використання логіки, були учені Роберт Ковальські (теоретичні основи), Маартен ван Емден (експериментальна демонстраційна система) з Единбургу і Ален Колмерое (реалізація) з Марселя. Популяризації мови Пролог багато в чому сприяла

ефективна реалізація цієї мови у середині 1970-х років Девідом Уорреном з Единбургу.

У 1977 р. Д. Уоррен і Ф. Перейра створюють в університеті Единбургу інтерпретатор/компілятор мови Пролог для ЕОМ DEC-10, тим самим, перевівши методи логічного програмування в практичну площину. У 1980 р. К. Кларк і Ф. Маккейб у Великобританії розробили версію Прологу для персональних ЕОМ. В 1984 р. було розроблено Пролог-компілятор для IBM-PC.

Пролог швидко надбав популярність в Європі як інструмент практичного програмування. У США ця мова в цілому була прийнята з невеликим запізненням у зв'язку з неефективною реалізацією мови логічного програмування Microplanner. Ідея чистої логіки привела у минулому до широкого розповсюдження невірних поглядів на мову Пролог. Вважалося, що на цій мові можна програмувати тільки міркування з висновком від цілей до фактів. Але істина полягає в тому, що Пролог – універсальна мова програмування і на ньому може бути реалізований будь-який алгоритм. Далека від реальності позиція «ортодоксальної школи» (послідовної) була подолана практиками мови Пролог, які прийняли прагматичніший підхід, скориставшись плідним об'єднанням нового, декларативного підходу з традиційним, процедурним.

У Японії навколо мови Пролог були зосереджені всі розробки комп'ютерів п'ятого покоління, метою якого декларувалося створення систем обробки інформації, що базуються на знаннях. З середини 90-х років XX ст. відбувся перехід Прологу від юності до зрілості. Зрілість мови означає те, що він більше не є науковою концепцією, яка ще визначається і уточнюється, а стає реальним об'єктом.

В цей же час з'являється багато комерційних реалізацій Прологу практично для всіх типів комп'ютерів. До найбільш відомих в Україні можна віднести Turbo Prolog.

За способами вибору мови програмування діляться на два класи. Пролог і його розширення, засновані на послідовному виконанні. Інші мови, такі, як

Parlog, Паралельний Пролог і GHC, засновані на паралельному виконанні. Послідовні мови відрізняються від паралельних методом реалізації недетермінізму. Відмінність Прологу від його версій полягає в методі вибору редукованої (спрощеної) мети.

Пролог – мова швидкого прототипування. Це означає, зокрема, що можна протягом якої-небудь години написати і відлагодити програму для експериментальної перевірки положень концепції. У теорії експертних систем це називається інкрементальним методом. Оскільки для досягнення високої якості роботи необхідне експериментування, то експертна система розвивається поступово. Такий еволюційний метод створення став домінуючим у області експертних систем.

Visual Prolog – продукт Датської фірми Prolog Development Center. Раніше розповсюджувався під назвою Turbo Prolog (Borland) і PDC Prolog.

Prolog Development Center витратив більше трьох років на розробку системи Visual Prolog з поетапним бета-тестуванням, в результаті одержали систему програмування, що відрізняється винятковою логічністю, простотою і ефективністю. Постачання комерційної версії почалися з лютого 1996. Visual Prolog є єдиною в світі Prolog-системою з 100% оболонкою, що виконана в ідеології Visual, та рядом високорівневих компонент, які спрощують розробку програм під управлінням сімейства ОС Windows.

Visual Prolog автоматизує побудову складних процедур і звільняє програміста від виконання тривіальних операцій. За допомогою Visual Prolog проектування призначеного для користувача інтерфейсу і пов’язаних з ним вікон, діалогів, меню, рядка повідомлень про стани тощо проводиться в графічному середовищі. Із створеними об’єктами відразу ж можуть працювати різні Кодові Експерти (Code Experts), які використовуються для генерації базового і розширеного кодів на мові Prolog, необхідних для забезпечення їх функціонування.

Потужність мови Visual Prolog в поєднанні з сучасною системою призначених для користувача інтерфейсів (GUI – Graphical User Interface)

робить простою і інтуїтивно зрозумілою розробку систем, що засновані на знаннях, системах підтримки ухвалення рішень, плануючих програмах, розвинених системах управління базами даних і т. п.

До новітніх досягнень належать засоби програмування на основі логіки обмежень (Constraint Logic Programming – CLP), які звичайно реалізуються у складі системи Prolog. Засоби CLP показали себе на практиці як виключно гнучкий інструмент для вирішення завдань складання розкладів і планування матеріально-технічного постачання.

Середовище візуальної розробки VDE (Visual Develop Environment) додатків системи Visual Prolog включає текстового редактора, різні редактори ресурсів, засоби розробки Help систем в гіпертекстовому уявленні, систему відстежування змін, яка забезпечує перекомпіляцію і перегенерацію тільки змінених ресурсів і модулів, ряд експертів коду, що оптимізує компілятор, і набір засобів проглядання різних типів інформації про проект. Повна інтеграція всіх засобів забезпечує підвищення швидкості розробки додатків. Одержані додатки є виконуваними .EXE програмами і не вимагають ніякого додаткового оточення і ліцензування.

Система програмування призначених для користувача інтерфейсів (VPI – Visual Programming Interface) системи Visual Prolog є високорівневою абстракцією властивостей, що підтримуються в сучасних операційних системах сімейства Windows. Це дозволяє легко здійснювати розробку сумісних додатків для цих платформ, і спрощує процес програмування в порівнянні з програмуванням в термінах вказаних операційних систем.

У систему включений також інтерфейс з базами даних, які підтримують мову структурованих записів SQL та Oracle. Майже всі типи баз даних доступні з використанням Windows ODBC (Open Database Connectivity) інтерфейсу.

Пізніше компанією Prolog Development Center були створені версії Visual Prolog 5.0, 5.1, 5.2, 6.0, 7.0 і 7.1, 7.2 і 7.3, остання 7.4, які функціонують в середовищах Windows, UNIX та клону Linux, OS для мобільних платформ. Варіант Personal Edition призначений для некомерційного використання.

У Visual Prolog реалізовані нові можливості логічного програмування:

- реалізована концепція об’єктно-орієнтованого програмування, що полегшує створення складних програмних систем;
- є обширні бібліотеки предикатів, що реалізовують математичні функції, засоби системного програмування, засоби для створення графічних інтерфейсів користувача тощо;
- інтегроване середовище розробки включає засоби візуального програмування;
- можливість створення і ефективної роботи з власними базами даних;
- засоби для роботи із зовнішніми базами даних, що мають різну архітектуру;
- засоби для створення розподілених додатків типу клієнт/сервер;
- можливість підключення програм, написаних на інших мовах (C, C++, асемблер і ін.).

Логічне програмування добре підходить для вирішення множини проблем:

- для роботи з формальними і природними мовами;
- для баз даних;
- для експертних систем;
- систем з реалізацією запитів;
- інших дискретних необчислювальних завдань.

Проте логічне програмування з використанням лише хорновських фраз, було б дуже вузьконаправленим. Тому використовуються і інші методи програмування. Деякі завдання по своєму характеру процедурні, і програмувати їх чисто декларативними мовами непрактично. Потрібні розвиненіші типи даних. Пролог і логічне програмування безперервно розширюються, охоплюючи все нові методи програмування і форми зображення саме у напрямі процедурного і об’єктно-орієнтованого програмування, а також у напрямі паралельних обчислень.

Логічне програмування і створені на його основі системи програмування, зокрема Turbo Prolog, Visual Prolog і ін., знаходять все більш широке застосування як інструментальний засіб для вирішення різних завдань з використанням ідей і методів теорії штучного інтелекту.

Логічне програмування дозволяє уникнути трудомісткої процедури представлення рішення задачі в алгоритмічній формі на мові програмування, як це робиться в процедурному програмуванні, та при рішенні ряду завдань дозволяє в десятки разів скоротити довжину програми в порівнянні з процедурним програмуванням і уникнути безпосередньої реалізації такої трудомісткої процедури як перебір з поверненням.

Мови логічного програмування характеризуються:

- високим рівнем;
- строгою орієнтацією на символьні обчислення;
- можливістю інверсних обчислень, тобто змінні в процедурах не діляться на вхідні і вихідні;
- можливою логічною неповнотою, оскільки часто неможливо виразити в програмі певні логічні співвідношення, а також неможливо одержати з програми всі висновки правильні.

Пролог – це високотехнологічна мова програмування, інтерес до якої знизився після невдачі проекту 5-го покоління ЕОМ, проте криза недовіри швидко минула і сьогодні вона популярніша, чим раніше. По критерію мінімуму кількості деталей реалізації, які потрібно тримати в голові при програмуванні «з листа» – Пролог не знає собі рівних. Проте логічні програми не відрізняються високою швидкодією, оскільки процес їх виконання зводиться до побудови прямих і зворотних ланцюжків міркувань різноманітними методами пошуку. Критикують Пролог в першу чергу за недостатню гнучкість, яка полягає в складності відладки програми, складної контрольованості проміжних результатів.

Prolog широко використовується в областях, для яких природно використання логічної парадигми:

- в символьних обчисленнях для рішення рівнянь, диференціювання та інтегрування;
- для створення систем календарного планування;
- для швидкої розробки прототипів прикладних програм;
- для створення адміністративних і Web-додатків;
- для проектування динамічних реляційних баз даних;
- у системах автоматизованого проектування;
- у системах обробки природничих мов;
- у дослідженнях штучного інтелекту.

Хоча спочатку Prolog було націлено на обробку природної мови, з тих пір він набув значного поширення в інших областях, як-то доведення теорем, експертні системи, ігри, системи автоматичних відповідей, онтології та складні системи керування. Сучасні середовища Прологу підтримують як створення графічних інтерфейсів користувача, так і адміністративні або мережеві застосування (організації сервера даних).

Застосування логічного програмування дозволяє швидко створювати прототипи систем різного призначення для експериментального дослідження і отримання якісних оцінок пропонованих рішень.

Visual Prolog може виконувати ті ж завдання, що і системи баз даних SQL, системи розробки програм C++, інші інструментальні засоби, такі як Visual Basic, Java, C#.

Запитання. Завдання

1. Мета створення мови програмування Prolog?
2. Сфера використання логіки в обчислювальній техніці?
3. Які мови можна віднести до мов логічного програмування?
4. Який з діалектів став основою стандарту ISO Prolog?
5. Які парадигми підтримує мова логічного програмування Prolog?
6. Яка основна теорія лежить в основі заснування мови Prolog?
7. На які обчислення орієнтовані мови логічного програмування?
8. Пояснити поняття «декларативні мови програмування».

9. Операційні системи, які підтримують реалізацію Prolog.
10. У якому році було створено мову Prolog?
11. Звернення до яких баз даних підтримується у Visual Prolog?
12. Нові можливості логічного програмування, які підтримує Visual Prolog?
13. Де використовується Visual Prolog?
14. Для вирішення яких проблем якнайбільше підходить мова Visual Prolog?
15. Недоліки мови програмування Visual Prolog?

Тест для самоконтролю знань[Ғ]

1. Які мови відносять до класу декларативних?
   а) функціональні і операторні;
   б) логічні і операторні;
   в) функціональні і логічні;
   г) аплікативні і операторні.
2. Декларативна мова програмування – це мова ...
   а) високого рівня, у якій не задається покроковий алгоритм рішення задачі;
   б) високого рівня, у якій реалізовано механізм пошуку з поверненням через оператори циклу;
   в) високого рівня, у якій реалізовано механізм обробки операторів;
   г) з покроковим алгоритмом рішення задачі.
3. Логічні мови програмування засновані на ...
   а) роботі зі списками;
   б) системі правил;
   в) зіставленні за зразком;
   г) всі відповіді вірні.
4. Функціональні мови програмування засновані на ...
   а) роботі з абстрактними обчислювальними моделями;
   б) обробці списків;
   в) роботі з сукупністю визначених функцій;
   г) всі відповіді вірні.
5. У чому полягає декларативна парадигма логічного програмування?
   а) передбачає використання математичної логіки для розробки програм;
   б) передбачає написання програм, які можуть змінювати свою власну поведінку;
   в) трактує процес обчислення як обчислення значень функцій в математичному розумінні, полягає в значенні, що повертається;

[Ғ] Усі запитання мають один правильний варіант відповіді.

г) визначає процес обчислень за допомогою логіки самого обчислення, а не логіки програми, яка управляє.

6. Яку з парадигм Visual Prolog підтримує на сучасному етапі?
   а) об'єктно-орієнтоване програмування;
   б) кероване подіями програмування;
   в) програмування з елементами імперативного стилю;
   г) всі відповіді вірні.
7. Мова Prolog заснована ...
   а) на властивому їй наборі операцій;
   б) на логіці диз'юнктів Хорна;
   в) на послідовності абстракцій та перетворень;
   г) всі відповіді вірні.
8. Логіка диз'юнктів Хорна – це ...
   а) підмножина логіки предикатів першого порядку, що сприймає в якості програми деякий опис завдання і проводить пошук рішення, користуючись механізмом бектрекінгу, уніфікації;
   б) підмножина логіки предикатів першого порядку, що сприймає в якості програми деякий опис завдання, виконує пошук рішення з використанням управляючих конструкцій If... then, Select Case;
   в) підмножина логіки предикатів першого порядку, що сприймає в якості програми деякий опис завдання, виконує пошук рішення з застосуванням циклічних структур For..., While та Until.
9. З використанням декларативного програмування …
   а) програма описує, який результат необхідно отримати, замість описання послідовності отримання цього результату;
   б) описується процес отримання результатів як послідовність інструкцій зміни стану програми.
10. Найпоширеніший діалект мови програмування Prolog?
    а) Strawberry Prolog;
    б) ISO Prolog;
    в) Edinburgh Prolog.
11. До недоліків Пролог відносять?
    а) строгу орієнтацію на символьні обчислення;
    б) складність контрольованості проміжних результатів;
    в) не можливість створення і ефективної роботи з власними базами даних.
12. Нові можливості Пролог?
    а) реалізація концепції об'єктно-орієнтованого програмування;
    б) метод реалізації недетермінізму;
    в) робота з формальними і природними мовами.
13. Сучасні середовища Прологу підтримують …
    а) адміністративні застосування;
    б) мережеві застосування;
    в) створення графічних інтерфейсів користувача;
    г) всі відповіді вірні.

14. Для яких операційних систем реалізований пролог?
    а) OS для мобільних платформ ;
    б) OS для мейнфреймів;
    в) сімейство Unix, Windows;
    г) всі відповіді вірні.
15. Для рішення яких завдань використовують засоби Constraint Logic Programming у системах Prolog?
    а) вирішення завдань складання розкладів;
    б) планування матеріально-технічного постачання;
    в) всі відповіді вірні.

РОЗДІЛ 1. ВІЗУАЛЬНЕ СЕРЕДОВИЩЕ РОЗРОБКИ VISUAL PROLOG

У розділі приведені системні, апаратні вимоги та рекомендації до установки Visual Prolog, розглянуті властивості візуального середовища розробки Visual Prolog 7.4.

1.1. Початок роботи

1.1.1. Системні вимоги

Для запуску візуального середовища розробки Visual Prolog (VDE – Visual Development Environment) на сучасному комп'ютері, необхідною умовою є наявність операційної системи з віконним інтерфейсом – Windows, Unix, Linux (KDE, Gnome та ін.), мінімум 32 Мбайта оперативної пам'яті та 100 Мбайт пам'яті для мінімальної установки, або 200 Мбайт для повної установки на жорсткому диску.

1.1.2. Установка Visual Prolog

Visual Prolog 7.4 Personal Edition можна безкоштовно завантажити за адресою http://www.visual-prolog.com/

Процедура інсталяції Visual Prolog практично однакова для всіх версій.

Варіант Personal Edition призначений для некомерційного використання, і повідомлення про це постійно є у всіх додатках, створених з його допомогою.

За замовчуванням установка Visual Prolog 7.4 не замінить всі раніше встановлені версії. Можна працювати з декількома версіями на одному комп'ютері.

- Комерційна версія буде за замовчуванням встановлена в
  C:\Program Files(x86)\Visual Prolog 7.4 або
  C: \ Program Files \ Visual Prolog 7.4.
- Personal Edition буде за умовчанням встановлена на
  C:\Program Files(x86)\Visual Prolog 7.4PE або
  C: \ Program Files \ Visual Prolog 7.4PE.

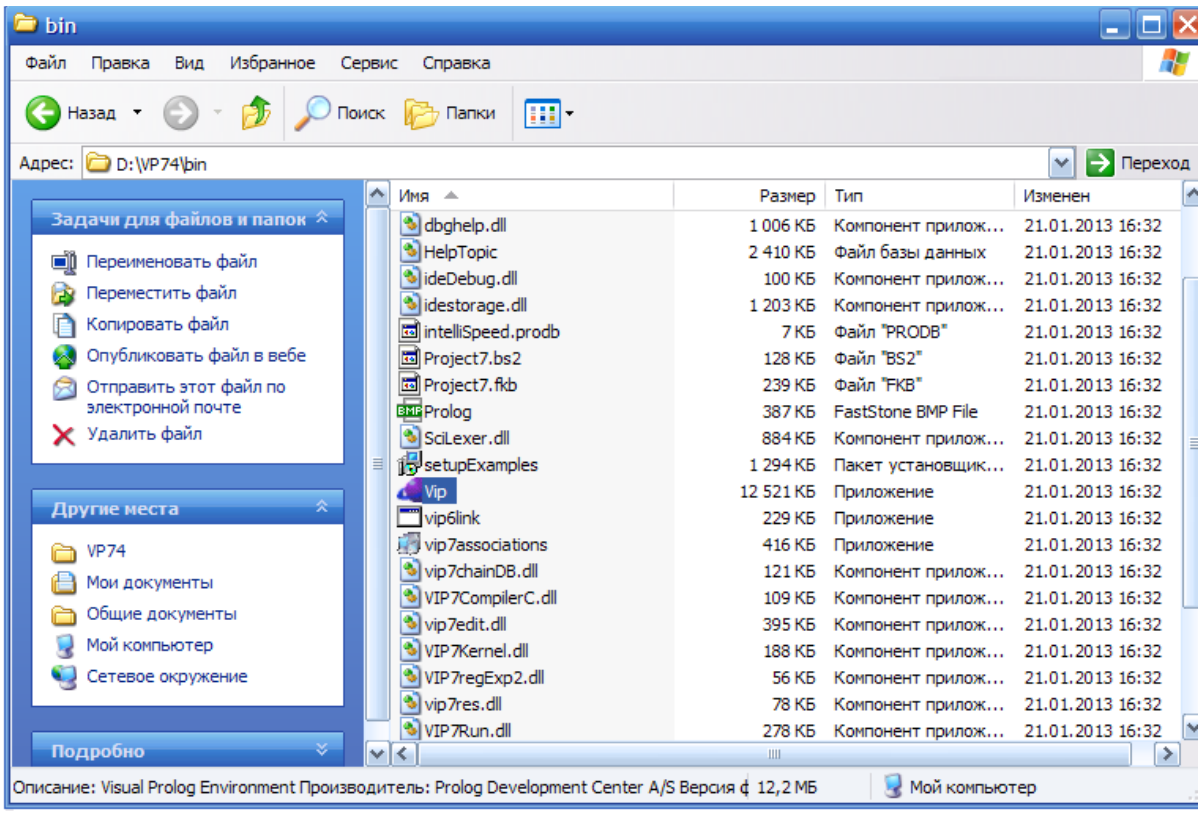


Рисунок 1.1. Завантаження Visual Prolog 7.4

Під час установки Visual Prolog слід встановити створення та  відображення ярлика в Головному меню та на Робочому столі для завантаження візуального середовища розробки.

Як варіант завантаження Visual Prolog з виконуючого файлу ..\BIN\Vip.EXE з основного каталогу Visual Prolog 7.4 (рис. 1.1).

1.1.3. Короткий опис вікна застосування

Проекти Visual Prolog 7.4 сумісні з проектам Visual Prolog 7.3. Якщо потрібно використовувати різні версії Visual Prolog, встановлені на одному комп'ютері, слід уникати відкриття проекту двічі (prj6 файлів). Деякі зміни можуть зажадати автоматичного оновлення в Visual Prolog 7.4 проектах. Тому, рекомендується зберегти файл проекту (PRJ6) до першої збірки. Крім того, в першу чергу рекомендується зробити копії всіх файлів проекту.

При відкритті проекту в Visual Prolog 7.4, IDE буде автоматично виконувати необхідні оновлення файлу проекту. Після цього рекомендується відновити весь проект за допомогою команди Build | Rebuild All і відповісти

«Так для всіх», щоб листи з пропозиціями додавання або видалення пакетів включити до заяви.

Visual Prolog 7.4 є повним середовищем програмування, що включає:

- Графічне інтегроване середовище розробки (IDE);
- Компілятор;
- Linker;
- Відладчик.

IDE Integrated Development Environment (українською – Інтегроване Середовище Розробки) – це середовище, яке використовується для створення, розробки та підтримки проектів Visual Prolog.

IDE використовується для рішення наступних задач:

- Створення (якщо проект створюється засобами IDE, то у момент створення обираються основні його властивості, такі як: чи являється проект виконуючим додатком чи DLL; використовує він користувацький інтерфейс чи заснований на текстовій взаємодії та ін.).
- Побудова (проект будується, тобто компілюється і зв’язується, використовуючи IDE).
- Перегляд (IDE і компілятор збирають інформацію про проект, яка використовується різними засобами для швидкої навігації по проекту тощо).
- Розробка (у процесі розробки і підтримки проекту IDE використовується для додавання і видалення з проекту файлів і елементів користувацьких інтерфейсів та їх редагування).
- Відладка (IDE використовується для відладки проекту. Відладчик використовується для переміщення по тексту програми та спостереження за станом ресурсів програми у процесі її виконання).

Під час завантаження IDE відображається стартова сторінка, як показано на рис. 1.2.

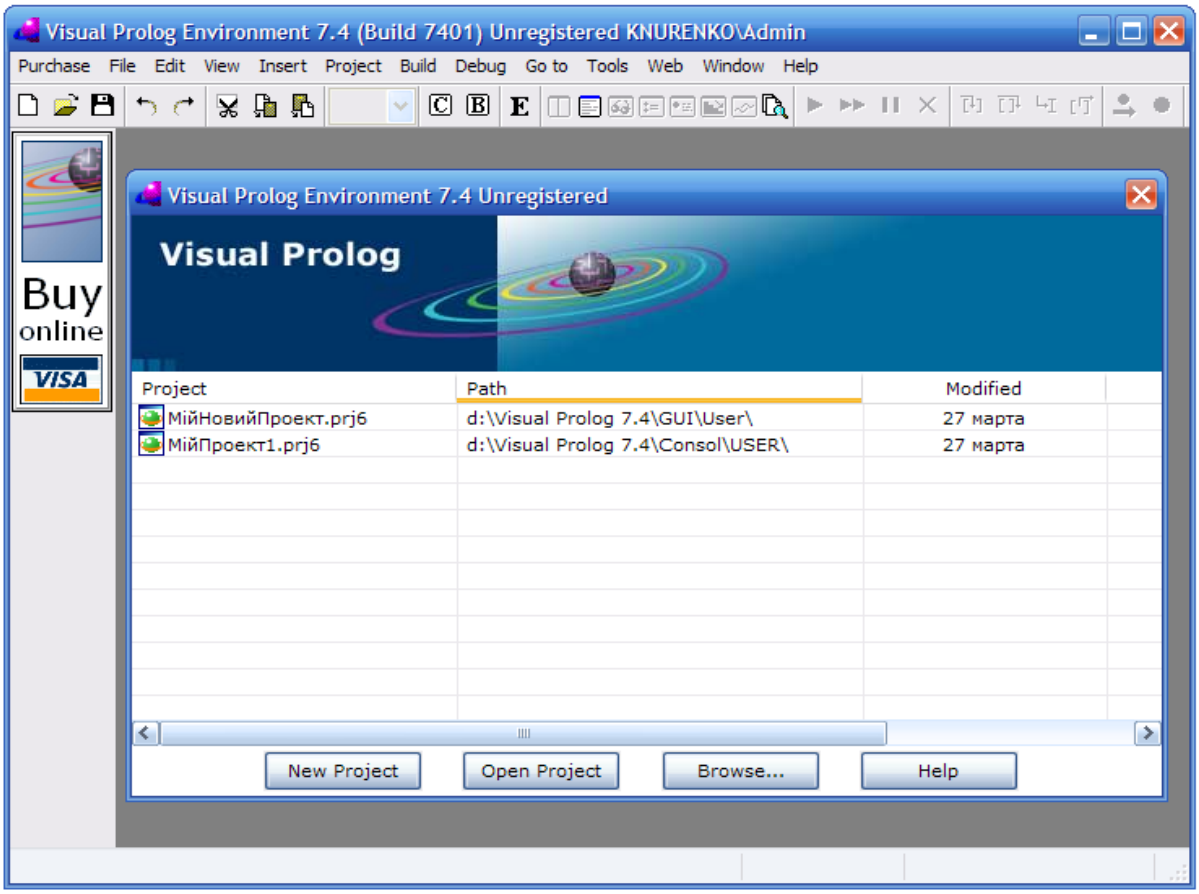


Рисунок 1.2. Вікно Visual Prolog 7.4. Стартова сторінка

У центрі може розміщуватися вікно запрошення Visual Prolog, яке містить список раніше реалізованих проектів та кнопки управління:

- New Progect – новий проект;
- Open Project – відкрити існуючий проект, виділений у списку;
- Browse – пошук та вибір каталогу для проекту;
- Help – довідка.

За допомогою цього вікна можна вибрати вже існуючий проект для роботи.

ICP має дві реалізації:

- Повна платна версія.
- Скорочена безкоштовна версія, в якій першим пунктом меню є Purchase (купити), що дозволяє за бажанням перейти до платної версії, як показано на рис. 1.3.

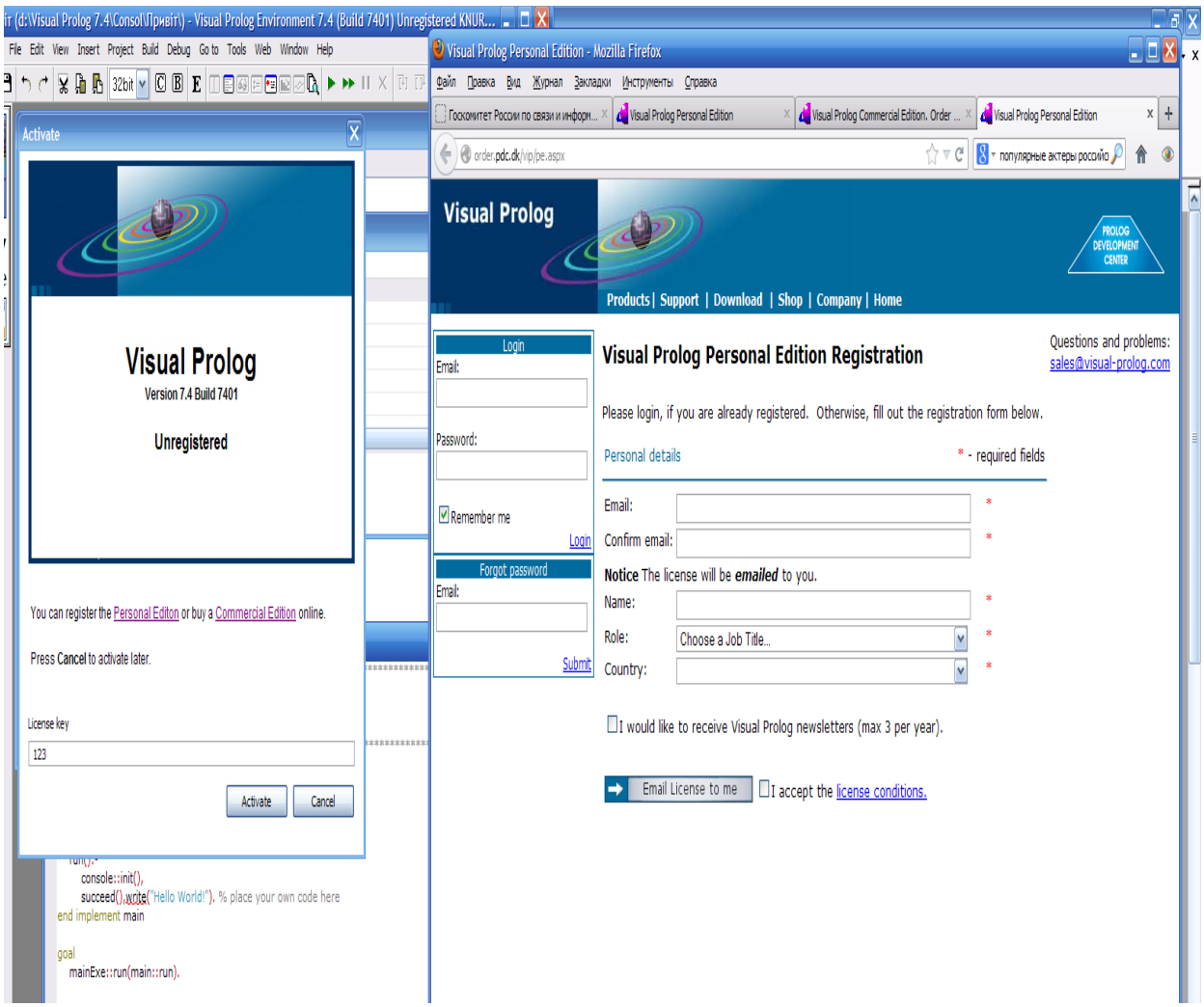


Рисунок 1.3. Придбати Visual Prolog Personal Edition 7.4

Головне вікно інтегрованого середовища розробки містить меню та панель інструментів (рис. 1.4).

Меню містить наступні пункти:

- File (файл) – операції с файлами (елементи цього меню дають змогу працювати з файлами – створювати, завантажувати, зберігати, друкувати; каталогами – змінювати, відображати; вийти із системи);
- Edit (правка) – редагування файлів (список команд цього меню надає можливість ефективно працювати з кодом програм (копіювати, вирізати, вставляти, видаляти, повторювати, відміняти команду, а також перемішатись у відповідний рядок у разі виникнення помилок та виконувати пошук);

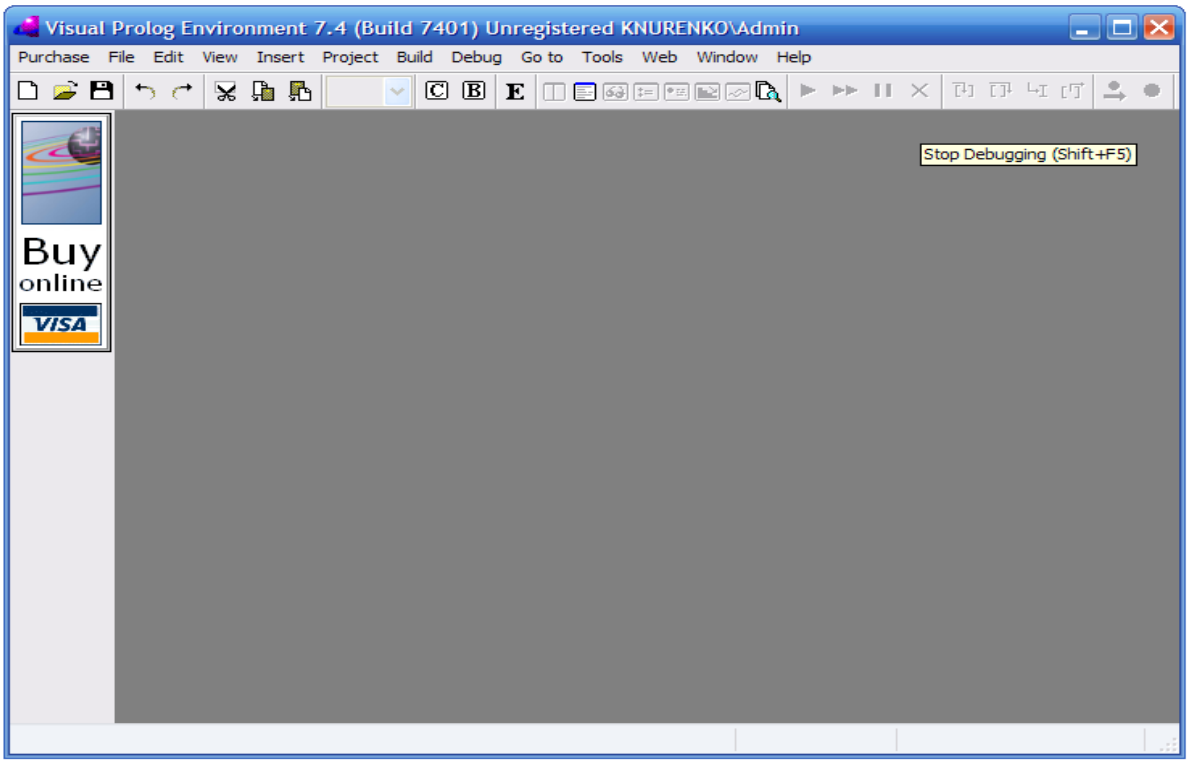


Рисунок 1.4. Головне вікно ICP Visual Prolog 7.4

- View (вид) – що відображати у вікні (команди меню дозволяєть відображати вікно проекту, повідомлень, ресурсів, текстовий редактор, експерт коду; вміст робочої пам'яті, додаткову інформацію про стан програми чи системи для відладки програми;
- Insert (вибрати) – що ввести у файл (дозволяє обирати ідентифікатори ресурсу та редактор коду, відображення оголошення класів, доменів, фраз предикатів і фактів, що використовуються у проекті, обрати шрифт);
- Project (проект) – операції з проектом (меню відображає команди роботи з об'єктами, такі як відкрити, створити, закрити, зберегти, обрати платформу);
- Build (побудувати) – побудувати проект (компілювати, виконати, завантажувати, зупинити, обрати платформу);
- Debug (відладка) – відладка (команди меню виконують завантаження та виконання проекту);

- Go to (перейти до) – вибір переміщення (меню містить команди переміщення по проекту за номером рядка, до оголошення, до визначення, до закладки тощо);
- Tools (інструменти) – вибір інструментального засобу (команди меню призначені для виконання збереження і передавання розширеної інформації про модулі пам'яті; доступ до сховищ з метою забезпечення програмним продуктом нової версії; вибирати змінні і їх значення для використання);

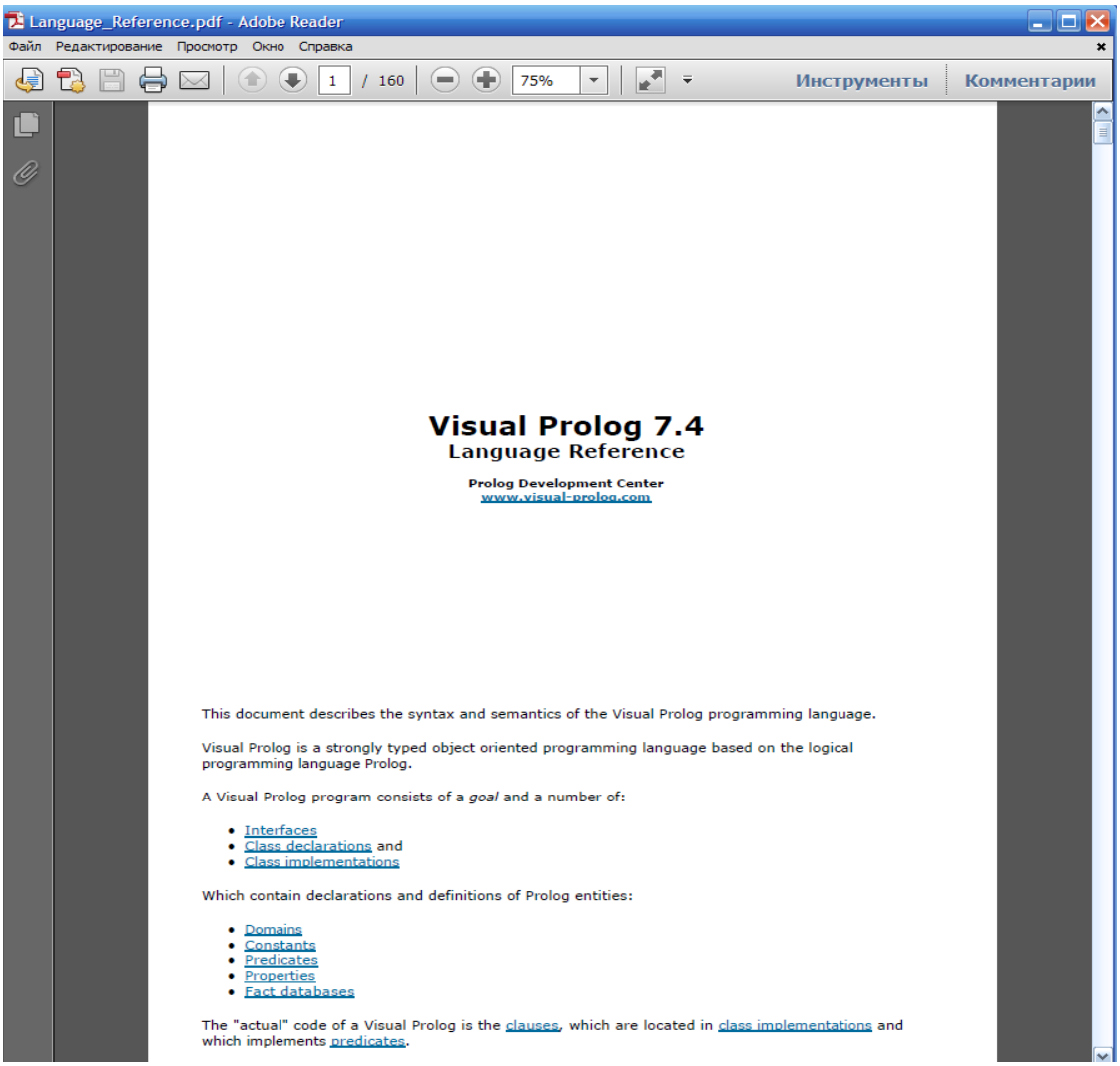


Рисунок 1.5. Підручник Language_Reference.pdf

- Web (Інтернет) – робота в Інтернеті (меню містить достатню кількість посилань на джерела інформації для швидкого опанування роботи у середовищі IDE Visual Prolog);
- Window (вікно) – відображення вікон;

- Help (довідка) – отримання довідки.

Для ознайомлення з роботою в Visual Prolog 7.4 рекомендується відкрити підручник – довідник з мови Prolog Development Center, вбудований в IDE (\Vip7.4\doc\Language_Reference.pdf). Цей документ (рис. 1.5) описує синтаксис і семантику декларативної мови програмування Пролог.

Пункт **меню File** містить команди роботи з файлами (рис 1.6).

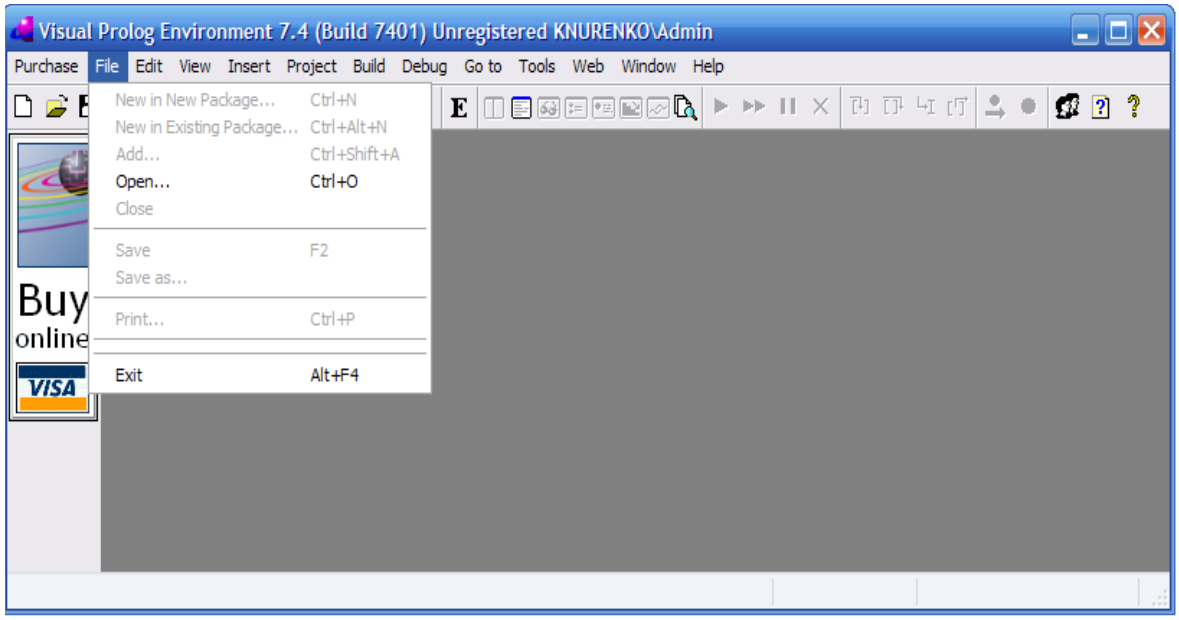


Рисунок 1.6. Меню Файл

Зміст **меню Файл** наступний:

- New in New Package… – створити нову тему у новому пакеті;
- New in Existing Package … – створити нову тему в існуючому пакеті;
- Add … – додати;
- Open – відкрити;
- Close – закрити;
- Save – зберегти;
- Save as … – зберегти як …;
- Print … – друк …;
- Exit – вихід з IDE.

Меню Edit (правити, рис. 1.7) містить наступні команди редагування:

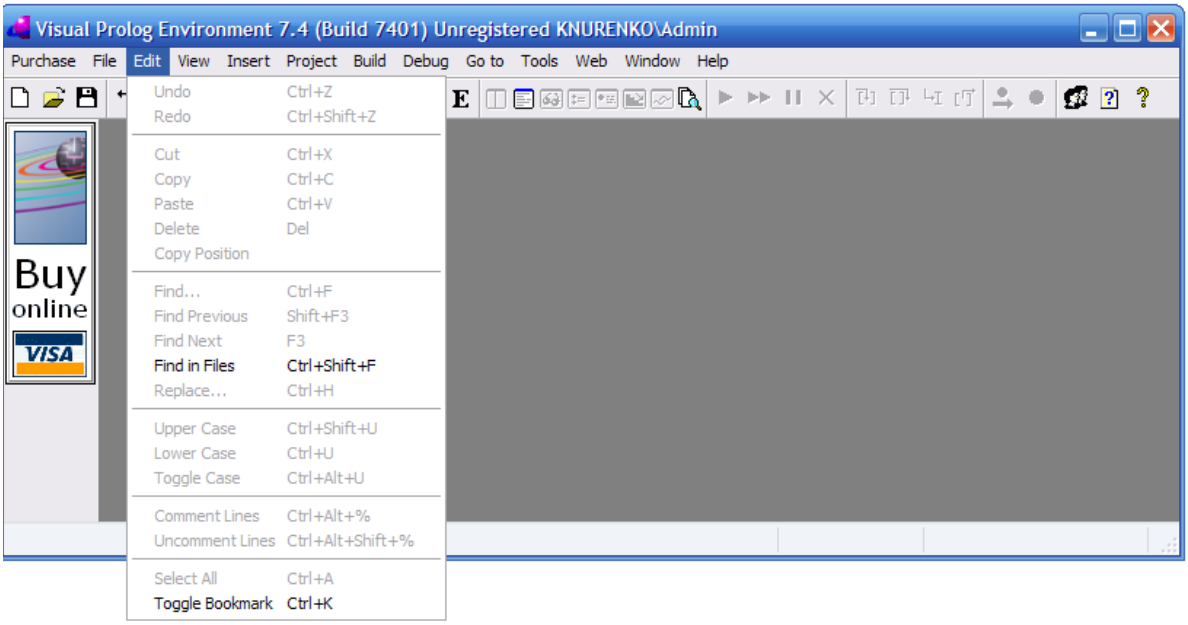


Рисунок 1.7. Меню Edit

- Undo – відмінити операцію;
- Redo – відновити операцію;
- Cut – вирізати виділене;
- Copy – копіювати виділене;
- Paste – вставити;
- Delete – видалити;
- Copy Position – копіювати позицію виділеного;
- Find … – знайти …;
- Find Previous – знайти з початку;
- Find Next … – знайти наступне …;
- Find in Files … – знайти у файлах …;
- Replace … – замінити виділене;
- Upper Case – верхній регістр;
- Lower Case – нижній регістр;
- Toggle Case – переключити регістр;
- Comment Lines – виконати виділені рядки коментарем;
- Uncomment Lines – виконати виділені рядки не коментарем;
- Select All – виділити все;

- Toggle Bookmark.

Пункт меню View (вид, рис. 1.8) містить наступні команди для перегляду:

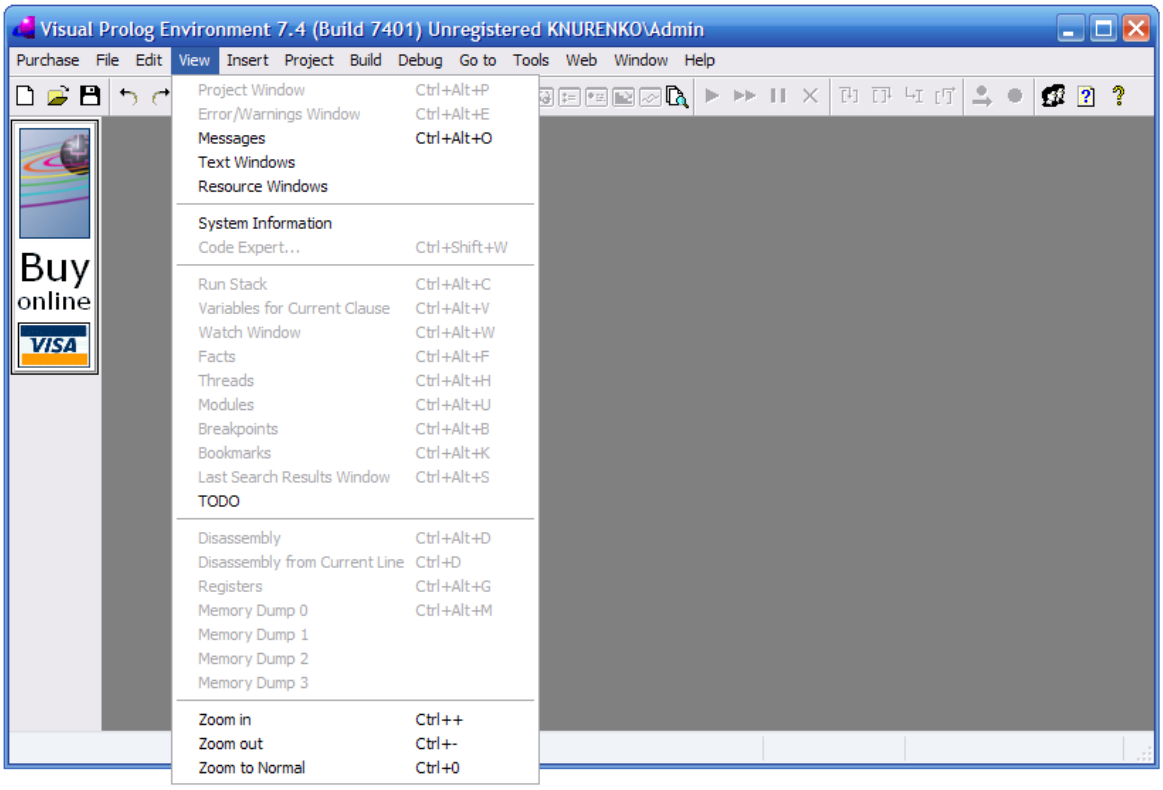


Рисунок 1.8. Меню View

- Project Window – вікно проекту;
- Error/Warnings Window – вікно помилок/попереджень;
- Message – повідомлення;
- Text Windows – текстові вікна;
- Resource Windows – вікна ресурсів;
- System Information – системна інформація;
- Code Expert … – Експерт коду;
- Run Stack – завантаження стека;
- Variables for Current Clause – змінні для поточної клаузи;
- Facts – факти;
- Threads – зв’язки;
- Modules – модулі;
- Breakpoints – точка зупинки;
- Bookmarks – закладки;

- Last Search Results Window – вікно результатів останнього пошуку;
- TODO – зробити;
- Disassembly – дизассемблер;
- Disassembly from Current Line – дизассемблер з поточного рядка;
- Registers – регістри;
- Memory Dump 0 – дамп пам’яті 0;
- Memory Dump 1 – дамп пам’яті 1;
- Memory Dump 2 – дамп пам’яті 2;
- Memory Dump 3 – дамп пам’яті 3;
- Zoom In – збільшити;
- Zoom Out – зменшити;
- Zoom Normal – нормальний масштаб.

Меню Insert(ввести, рис. 1.9**)** містить команди введення у файл компонент:

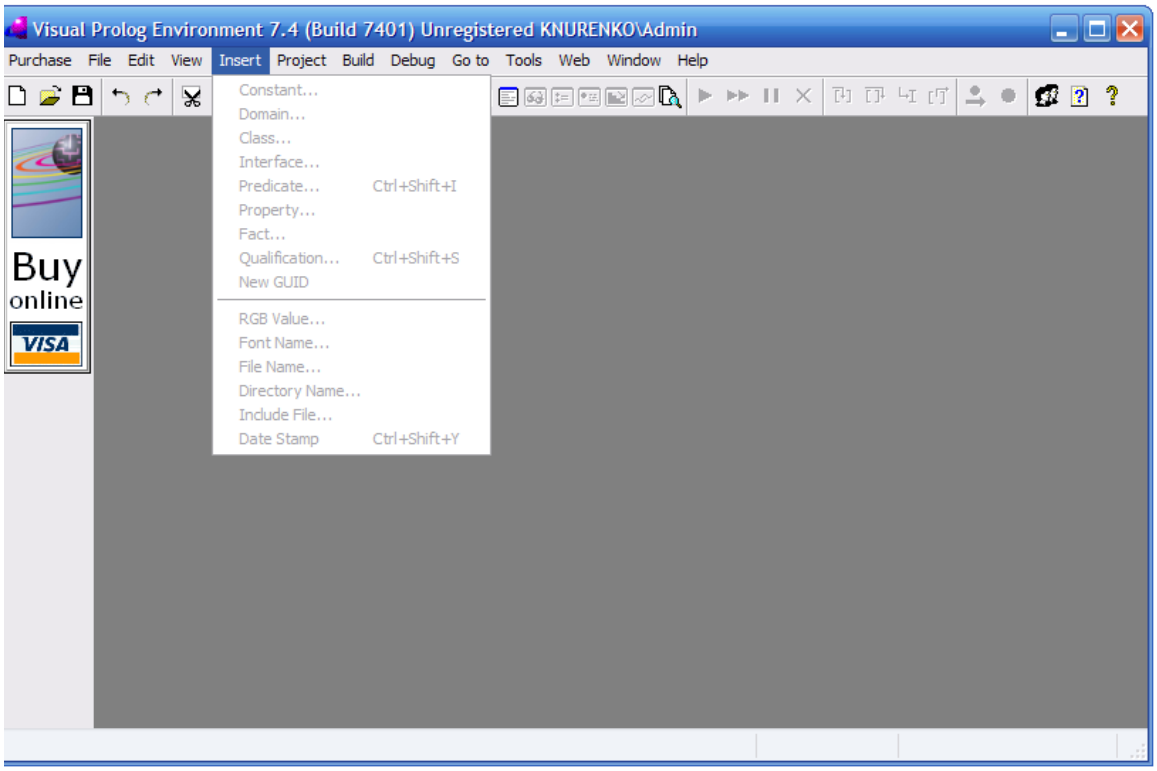


Рисунок 1.9. Меню Insert

- Constant … – константа;
- Domain … – домен;

- Class … – клас;
- Interface … – інтерфейс;
- Predicate … – предикат;
- Property … – властивості;
- Fact … – факт;
- Qualification … – кваліфікація;
- New GUID – новий GUID;
- RGB Value … – значення RGB;
- Font Name … – ім’я шрифту;
- File Name … – ім’я файлу;
- Directory Name … – ім’я каталогу;
- Include File … – включити файл;
- Date Stamp – дата.

Project (проект**)**

Пункт Project (рис. 1.10) містить наступні команди роботи з проектом:

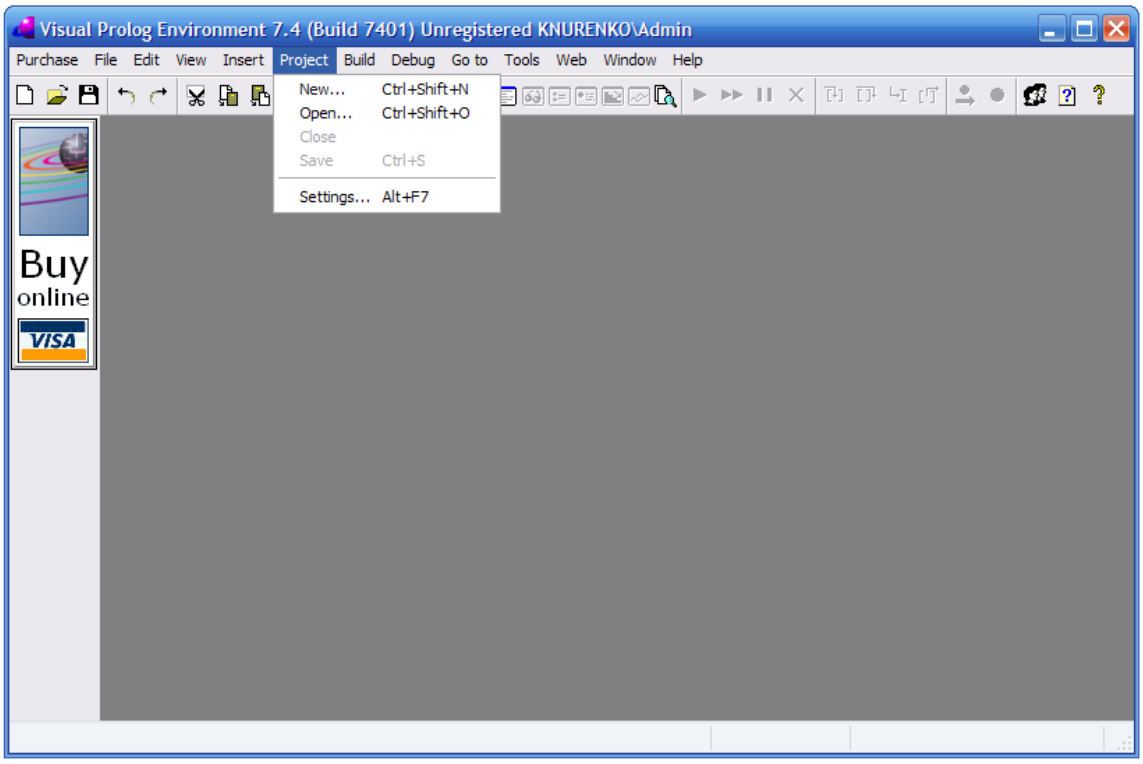


Рисунок 1.10. Меню Project

- New … – створення нового проекту;
- Open … – відкрити збережений проект;

- Close – закрити поточний проект;
- Save – зберегти поточний проект;
- Setting … – налаштування.

Build (побудувати**)**

Містить наступні команди побудови проекту (рис. 1.11):

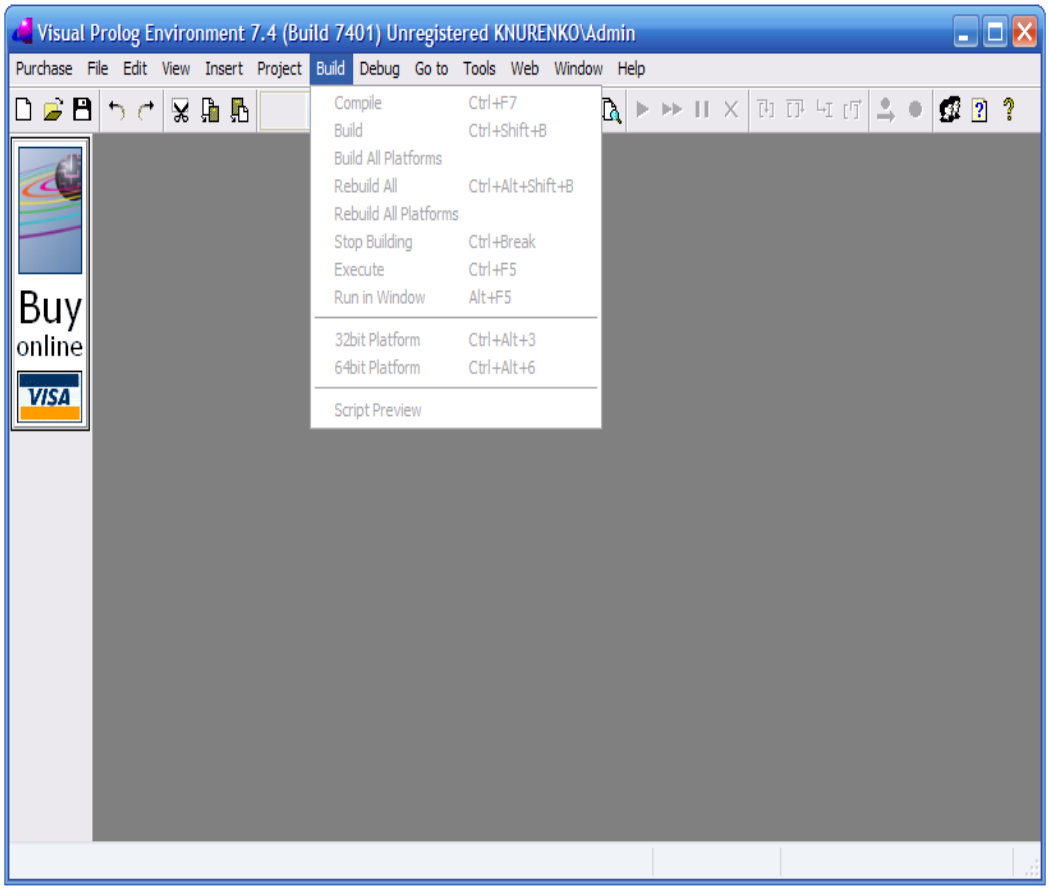


Рисунок 1.11. Меню Build

- Compile – компілювати;
- Build – побудувати;
- Build All Platforms – побудова для всіх платформ;
- Rebuild All – перебудувати все;
- Rebuild All Platforms – перебудувати для всіх платформ;
- Stop Building – зупинити побудову;
- Execute – завантажити;
- Run in Windows – завантажити у Windows;
- 32bit Platform –
- 64bit Platform –

- Script Preview – попередній перегляд сценарію.

Debug (відладка**)**

Містить команди налаштування проекту (рис. 1.12):

- Run – завантаження;
- Run Skipping Soft Breakpoint – виконати до точки переривання;
- Stop Debugging – зупинити налаштування;
- Break Program – вийти з програми;
- Restart – повторне завантаження;
- Attach Process – приєднати процес;
- Step Over – крок без зупинки у процедурі;
- Step Info – крок з входом до процедури;
- Step Out Of – крок виходу з процедури;
- Run to Cursor – виконання до позиції курсору;

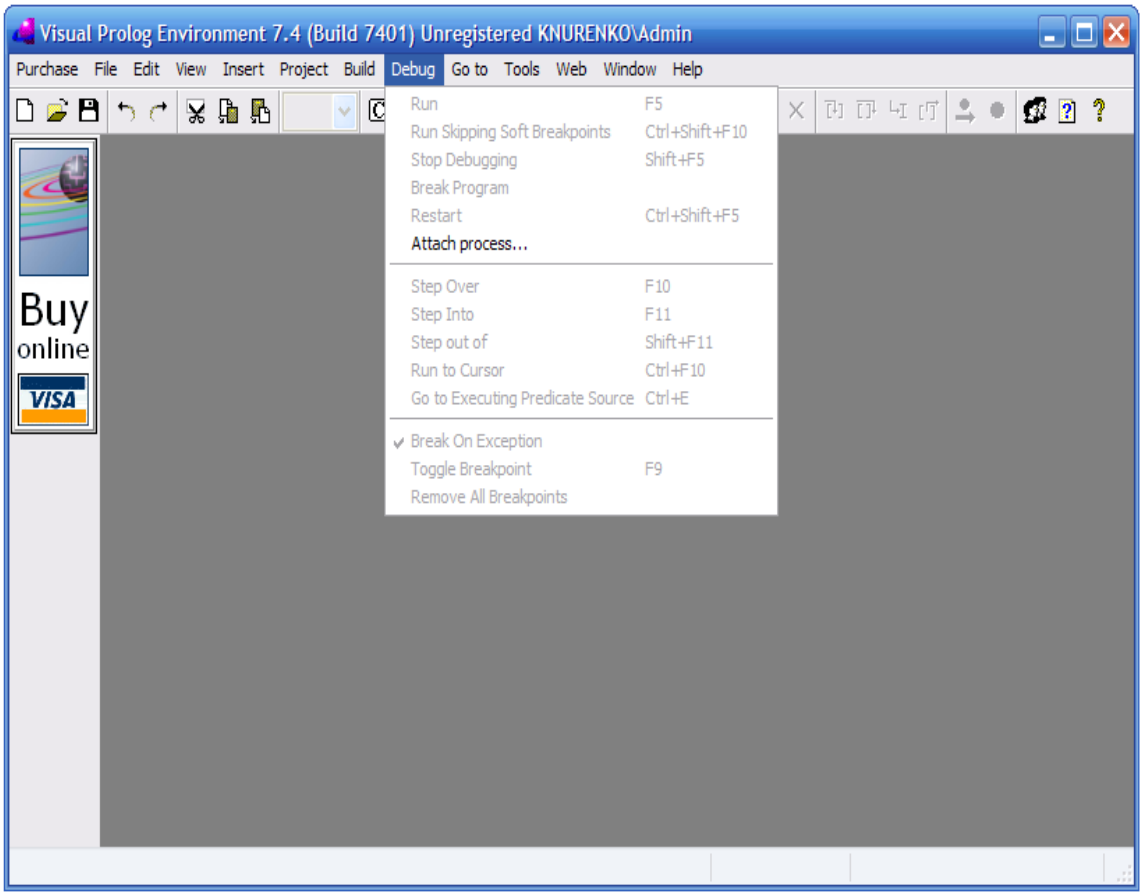


Рисунок 1.12. Меню Debug

- Go to Executing Predicate Source – перейти до вхідного предикату запуску;
- Break on Exception – вихід за виключенням;
- Toggle Breakpoint – переключити точку переривання;
- Remove All Breakpoint – видалити всі точки переривання.

Go To (прейти до**)**

Містить наступні команди переміщення по проекту (рис. 1.13):

- Source Browser … – браузер вхідного коду;
- Go to Declaration – йти до оголошення;
- Go to Definition – йти до визначення;
- Go to Related Files … – йти до зв'язаних файлів;
- Go to Line Number … – йти до рядка за номером;
- Go to Position on Clipboard – йти до позиції у буфері обміну;
- Go to Next Bookmark – перейти до наступної закладки;

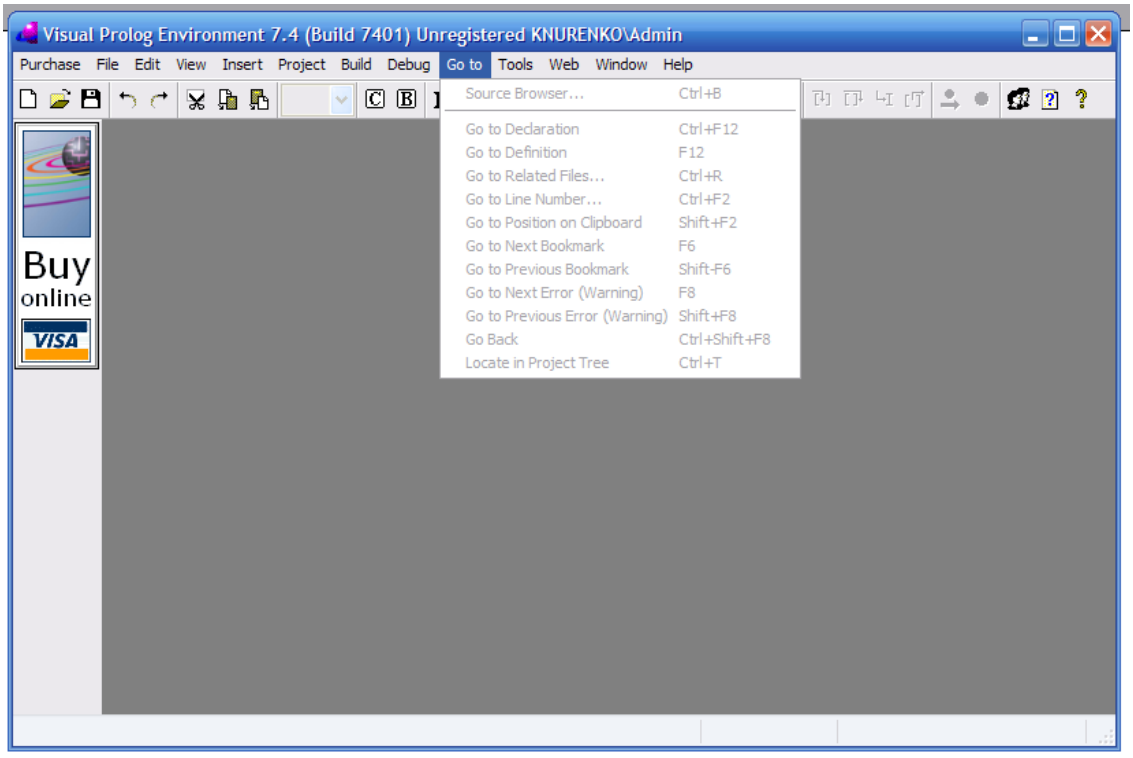


Рисунок 1.13. Пункт меню Go To

- Go to Previous Bookmark – перейти до попередньої закладки;
- Go to Next Error (Warning) – йти до наступної помилки (попередження);

- Go to Previous Error (Warning) – йти до попередньої помилки (попередження);
- Go Back – повернутися назад;
- Locate in Project Tree – локалізація у дереві проекту.

Tools (інструменти**)**

Містить команди вибору інструментальних засобів (рис. 1.14):

- Memory Profiler … – профілі пам'яті;
- Configure Tools … – засоби конфігурування;
- Source Control Repository … – репозиторій управління вхідним кодом;
- IDE Variables … – змінні IDE;
- Options … – опції.

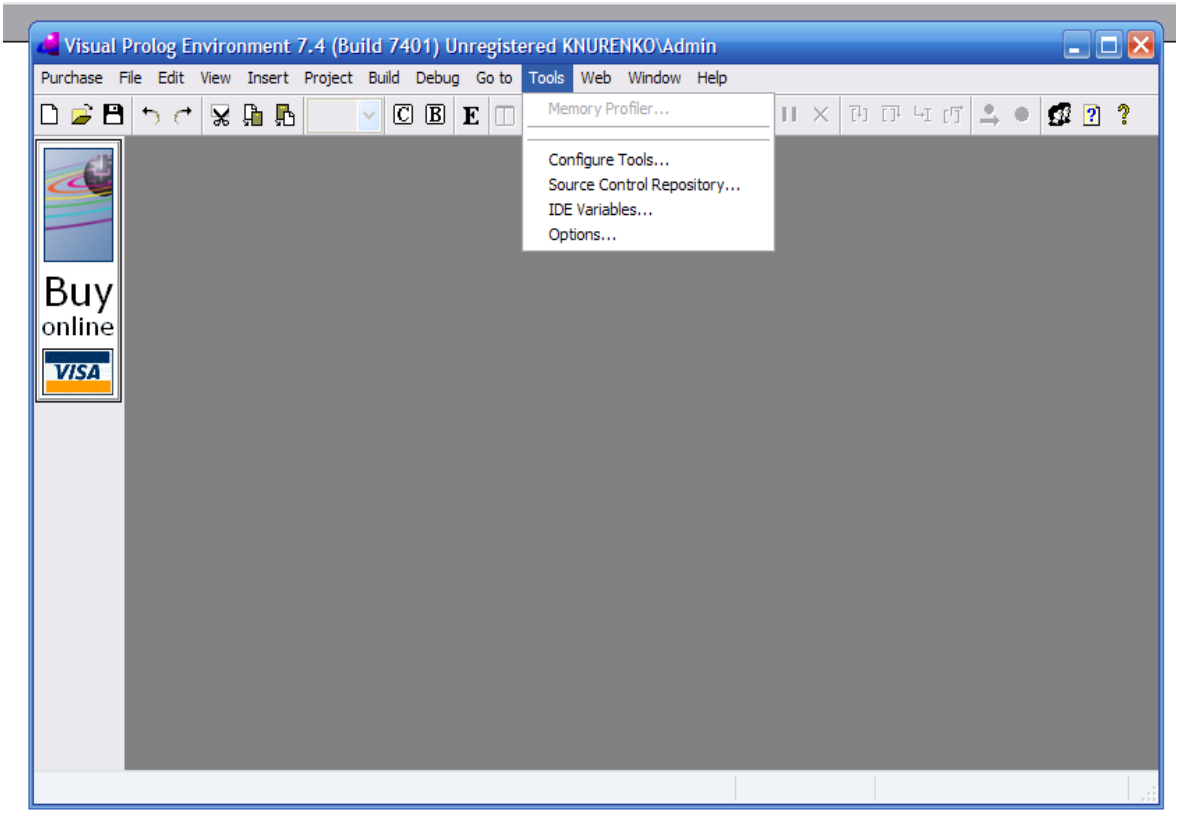


Рисунок 1.14. Меню Tools

Web (Інтернет**)** містить наступні команди доступу до Інтернет (рис. 1.15):

- Tutorials – підручники;
- Wiki – Wiki;
- Discussion Forum – Форум обговорень;

- Check for Updates – перевірити поновлення;
- Knowledge Base – база знань;
- My Registration – Моя реєстрація;
- Visual Prolog – Visual Prolog;
- Prolog Development Center – центр розробки Prolog.

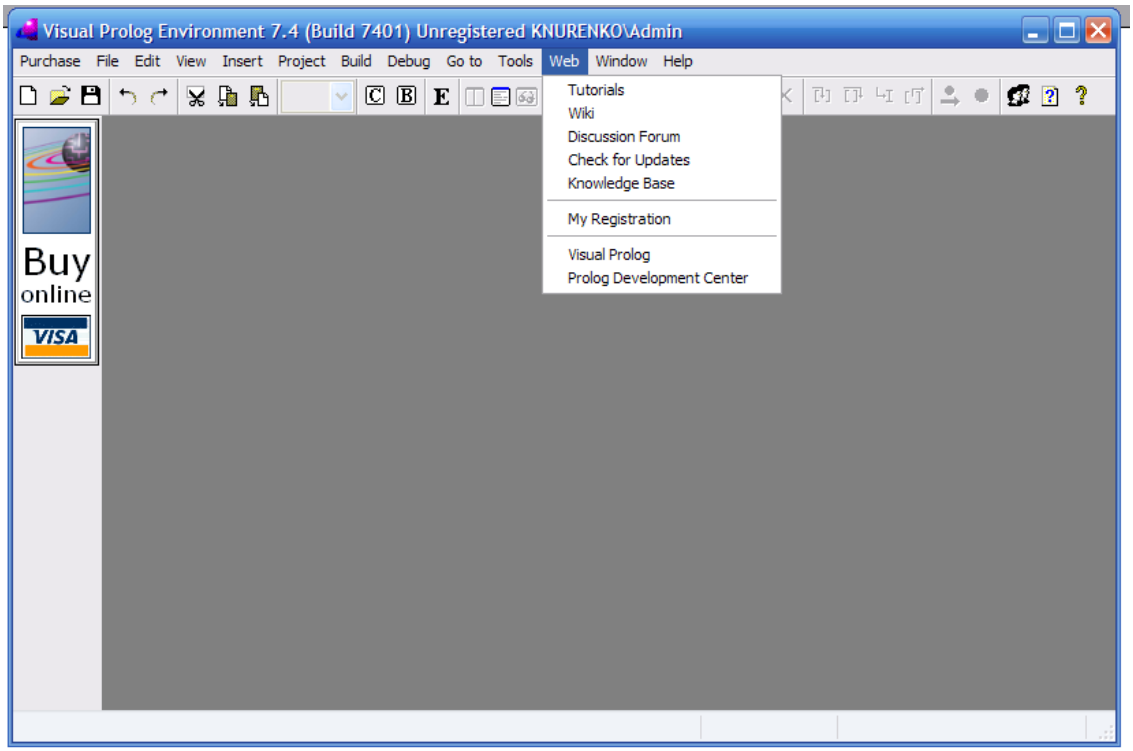


Рисунок 1.15. Меню Web

Window (вікно**)**

Містить такі команди вибору відображення вікна (рис. 1.16):

- Cascade – каскадне;
- Horizontal Tile – поряд по горизонталі;
- Vertical Tile – поряд по вертикалі;
- Arrange Icons – розміщення іконок;
- Lock All Editor Windows together – зв’язати всі вікна редакторів;
- Close All Editor Windows Expert the Active – закрити всі вікна редакторів, окрім активного;
- Prune Editor Windows – вікна чернеток редакторів.

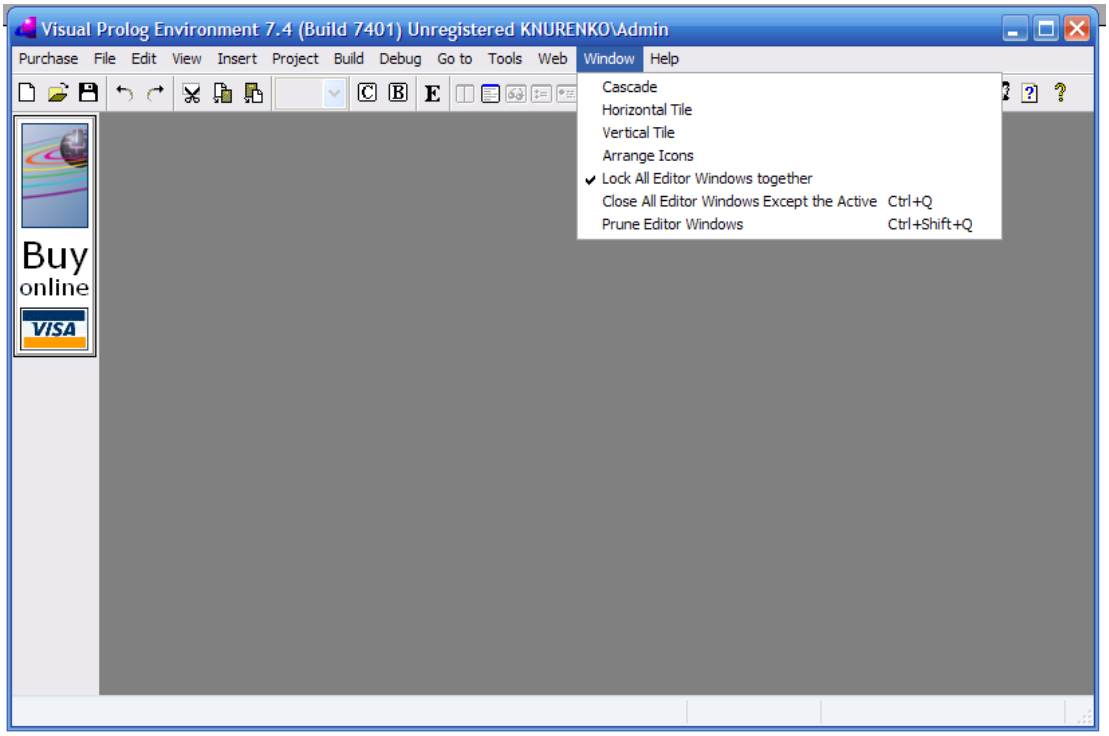


Рисунок 1.16. Меню Window

Help (довідка**)**

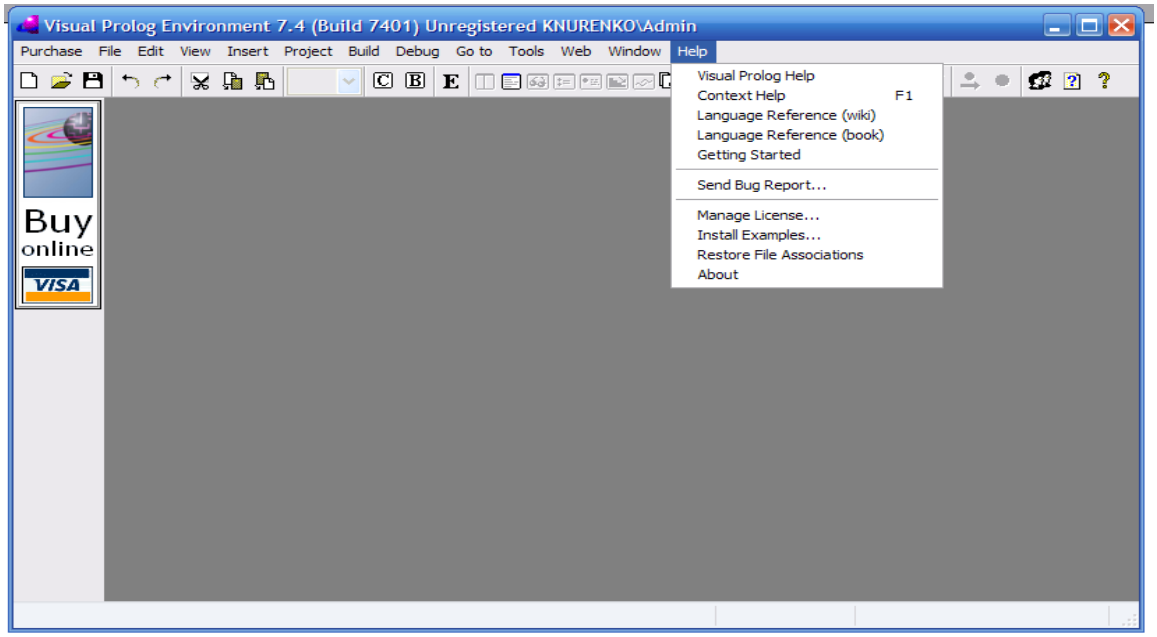


Рисунок 1.17. Меню Help

Містить засоби отримання довідки (рис. 1.17):

- Visual Prolog Help – довідка Visual Prolog;
- Context Help – контекстна довідка;
- Language Reference (wiki) – посилання (wiki);
- Language Reference (book) – посилання (book);

- Getting Started – навчання для початківців;
- Send Bug Report … – відправити звіт про помилки;
- Manage License … – отримати ліцензію;
- Install Examples … – встановити приклади;
- Restore File Associations – перестановити IDE;
- About – про розробника IDE (рис. 1.18).

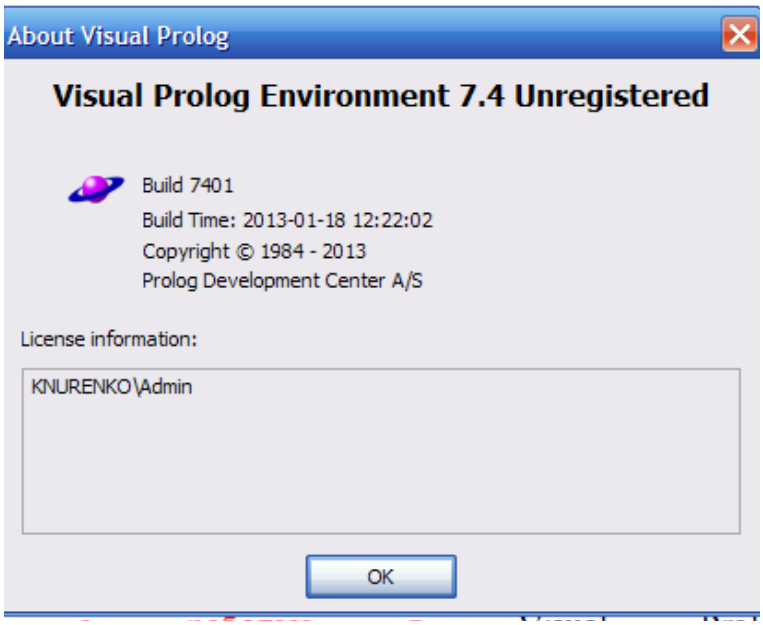


Рисунок 1.18. Довідка: інформація про розробника IDE

Довідка містить пункт Getting Started, який розробники у першу чергу рекомендують прочитати новачкам у Visual Prolog. І незалежно від того, чи є ви досвідченим користувачем Visual Prolog або новачок, ви знайдете багато цікавого в Visual Prolog з інтернет-ресурсів, доступних безпосередньо з інтегрованого середовища розробки (IDE) Visual Prolog за допомогою пункту меню Інтернет або відвідавши сайт Visual Prolog. Для вивчення Visual Prolog 7.4 меню Help вікна застосування пропонує встановити приклади (Install Examples).

Вбудований підручник IDE Visual Prolog

Інтегроване середовище розробки Visual Prolog містить багато класів, переглянути які можна використовуючи вбудований підручник з меню Help (довідка), як показано на рис. 1.19.

Вікно містить дві вкладки: зміст довідки та інформація по вибраному розділу.

Довідка Visual Prolog пропонує розділи:

- Integrated Development Environment (IDE) – Інтегроване Середовище Розробки;
- Prolog Foundation Classes (PFC) – базові класи мови Prolog.

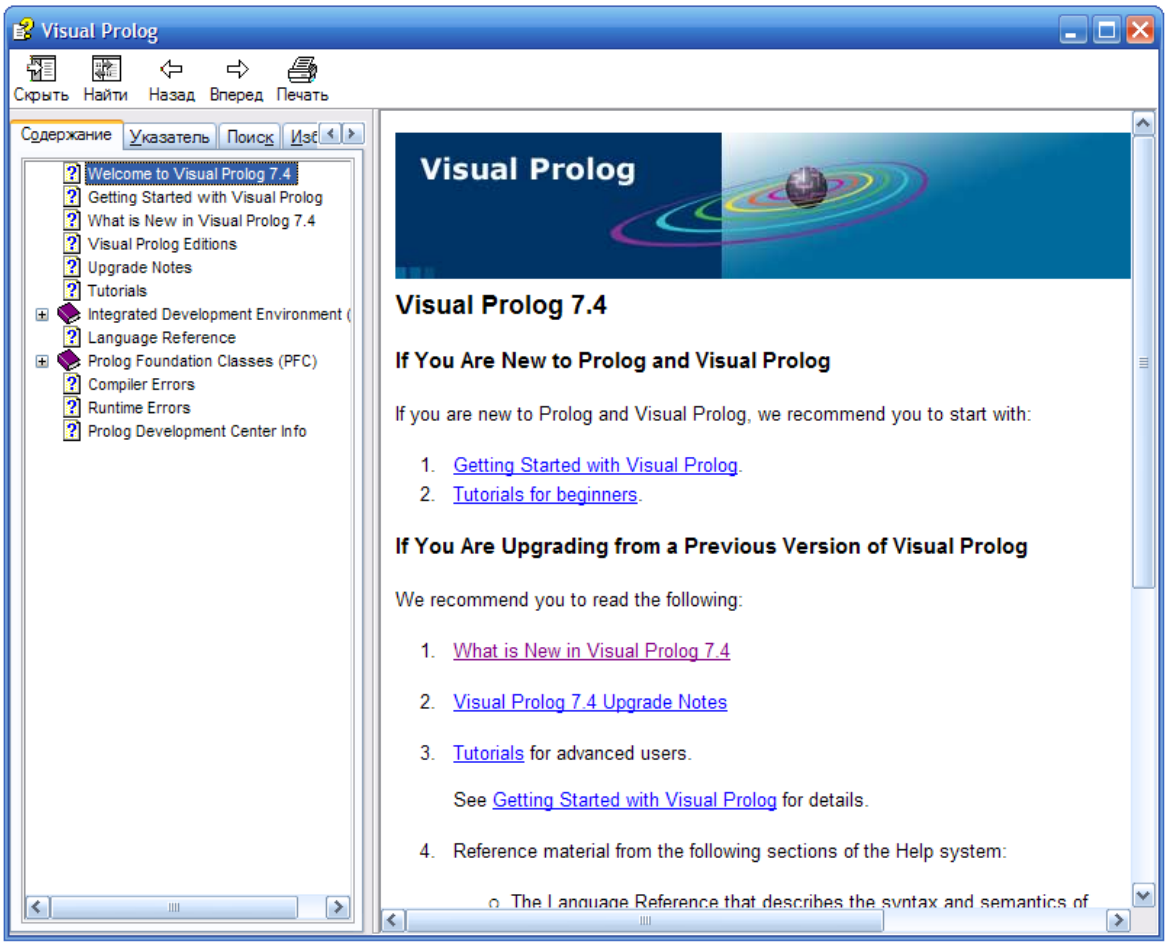


Рисунок 1.19. Вікно перегляду довідкової інформації

1.2. Інтегроване Середовище Розробки (IDE)

Visual Prolog 7.4 – це нове покоління логічної мови програмування Visual Prolog, яка може бути використана для створення розробок промислового застосування для платформи Microsoft Windows (Windows 7, Windows Vista, Windows XP, Windows Server 2008, Windows Server 2003).

Visual Prolog (VIP) складається з кількох елементів. Він має інтерактивне інтегроване середовище розробки (IDE), яке включає в себе текст і різні

редактори ресурсів, генерації коду і коду перегляду інструментів (Код експертів), побудову логіки управління (об'єкт) і т. д. IDE використовує різні інструменти командного рядка: командний рядок Visual Prolog компілятор, командний рядок Visual Prolog (або Microsoft) лінкер (рис. 1.20).

Інтегроване середовище розробки розроблене для зручної і швидкої розробки, тестування і зміни додатків, написаних на Visual Prolog, підтримує 32- та 64-розрядні ОС Windows на платформах MS Windows XP/Vista/Windows7 на базі процесорів Intel, сумісних за архітектурою.

Слід розпочинати розробку нових проектів від налаштування IDE в проекті. Тут потрібно вказати ім'я для проекту, тип проекту і теку. Проект створюється за замовчуванням утилітою *Параметри проекту*.

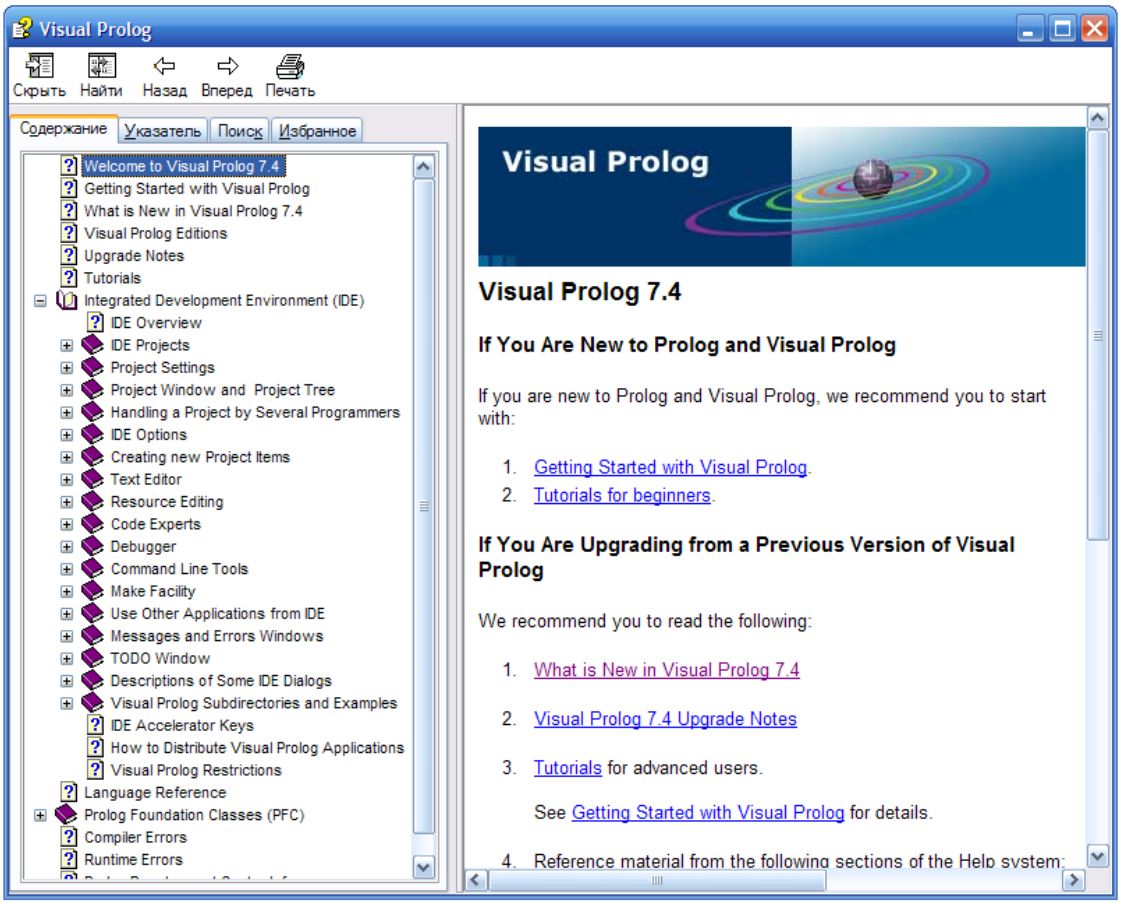


Рисунок 1.20. Розділ довідки з IDE

Після того, як проект буде побудований, ***дерево*** проекту відображає всі файли, створені в проекті і стандартні файли, використовувані в проекті.

Довідка надає інформацію про деякі бібліотеки імпорту та деякі стандартні пакети Visual Prolog Foundation Classes (PFC).

IDE надає Код експертів, Редактор ресурсів, Текстовий редактор, Код браузера і інші корисні інструменти, які допомагають розвивати проект. Редактори ресурсів використовуються для визначення, верстки та редагування вікон, форм, діалогів, IDE управління, меню, панелей інструментів, іконок, курсорів і растрових зображень.

За запитом програміста, Експерти Коду можуть викликатися у будь-якій частині генерованої бази у вікні редактора для перевірки з логікою програми та заповнення. Редагування коду в текстовому редакторі IDE має додаткові сервісні функції, часто використовувані функції будуть доступні клацанням правої кнопки миші по багаторівневому спливаючому меню редагування, вибору, пошуку, заміни і вставки функцій. Також код може бути точно виконаний вставкою команд з підменю Insert.

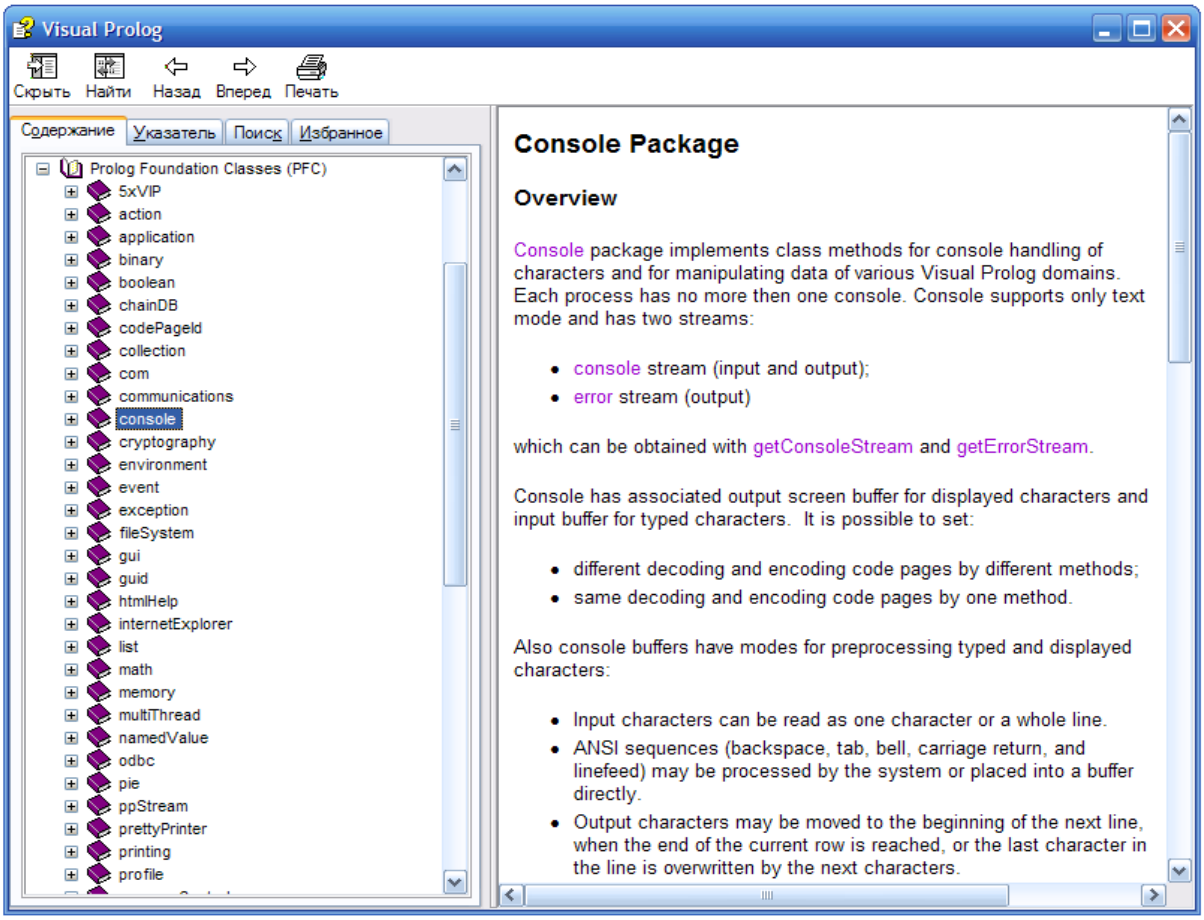


Рисунок 1.21. Розділ довідки FPC

Базові класи мови Prolog (PFC) – це набір базових класів мови Прологу в IDE, переглянути які можна у вікні довідки, як показано на рис. 1.21.

Prolog Foundation Classes (PFC) є набором класів і інтерфейсів, які складають основу для написання різних видів додатків і програмних компонентів в Visual Prolog. Класи та інтерфейси організовані в ієрархії пакетів. Пакет являє собою групу тісно пов'язаних інтерфейсів і класів, тому користувачам зручно їх описувати в одному місці.

Важливим є те, щоб включати у призначених для користувача програмах, файли (*. PH файли) в заголовок пакету, а не для окремих класів (CL-файли) і інтерфейсів (I-файли). Причини для такого включення наступні: PH-файли містять директиви, які обробляються VDE. На основі необхідних пакетів, зазначених в цих директивах, вимагається, щоб всі пакети, необхідні для обробки, були включені в проект.

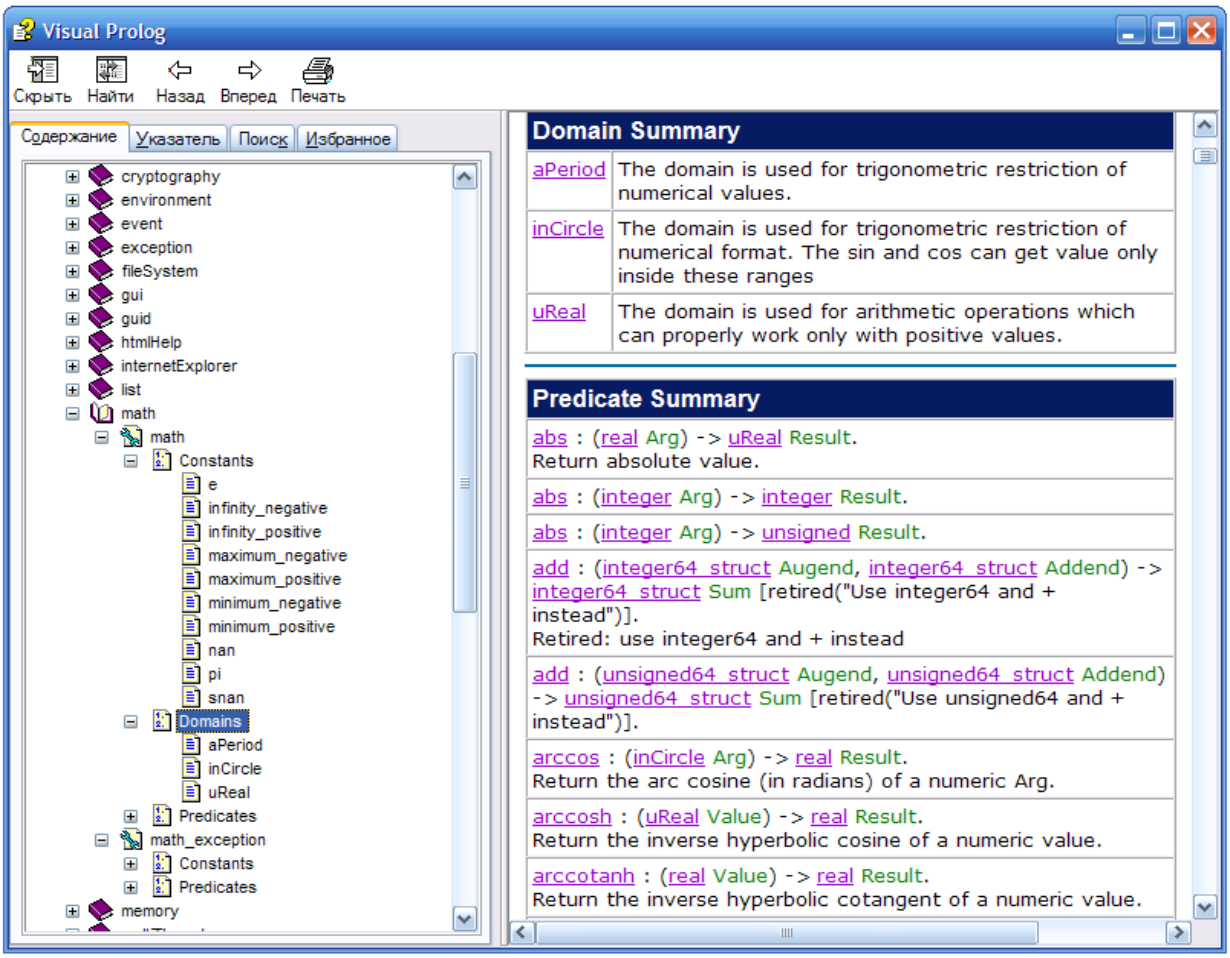


Рисунок 1.22. Довідка. Об'єкти класу math

Клас, який знову використовується, може посилатися на деякі додаткові класи. PH-файл містить всі необхідні директиви. Клас *ядро* містить області і константи, які зазвичай використовуються в класах PFC і інтерфейсів.

Для кожного класу представлені його об'єкти. Наприклад, для класу *math*, як показано на рис. 1.22.

Пакет математика підтримує математичні операції Visual Prolog з точки зору цифрової області: ціле, беззнакове чи дійсне число.

1.2.1. Структура програми

Для того, щоб зрозуміти принцип роботи Visual Prolog, слід розглянути традиційну програму «*Привіт*» (рис. 1.23) та розібрати кожен рядок її коду.

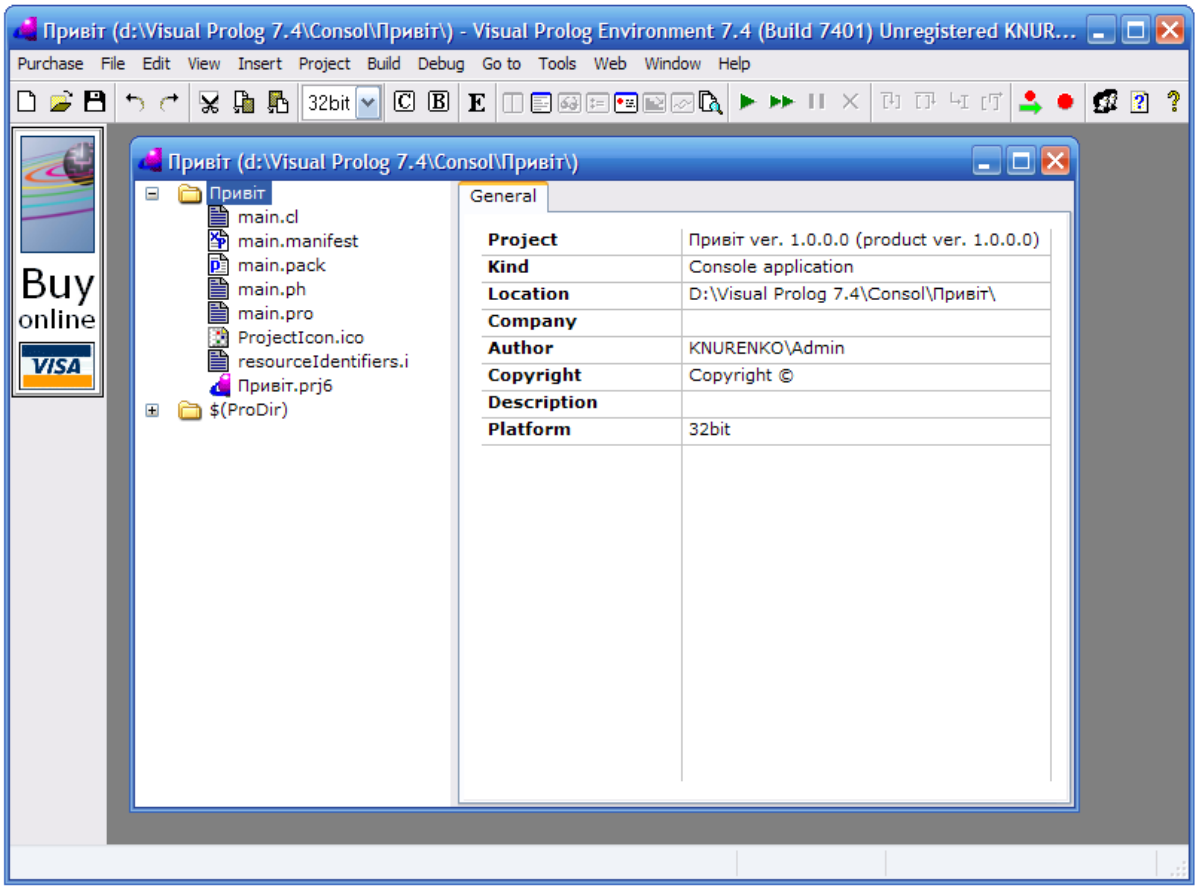


Рисунок 1.23. Проект «Привіт»

Проект *Привіт.prj6* включає модулі, кожен з яких зберігається в окремому файлі:

- *Привіт.prj6* – містить правила формування проекту;
- *main.cl* – оголошення головного класу. Для консольного додатку містить посилання на предикат *run* (рис. 1.24);

- *main.manifest* – містить інформацію про системні засоби побудови проекту;
- *main.pack* – пакет головного класу, містить посилання на модулі проекту;
- *main.ph* – містить імена файлів проекту;
- *main.pro* – реалізація головного класу, містить виконуючий код;
- *resourcedentifiers.i* – файл інтерфейсу класу.

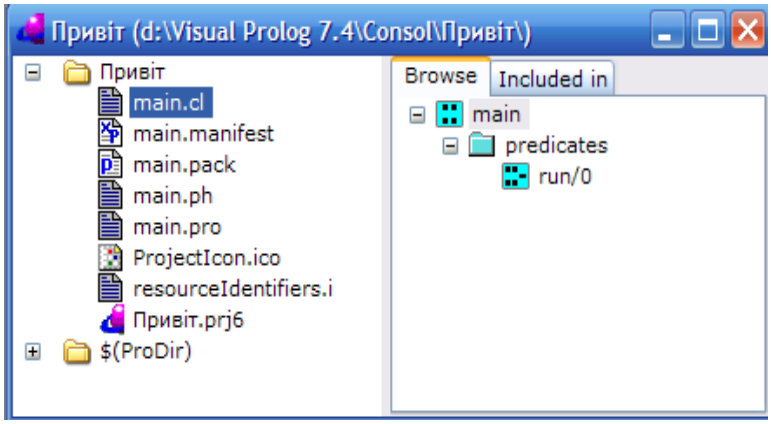


Рисунок 1.24. Файл проекту main.cl

Лістинг модуля оголошення класу main.cl наступний (рис. 1.25):

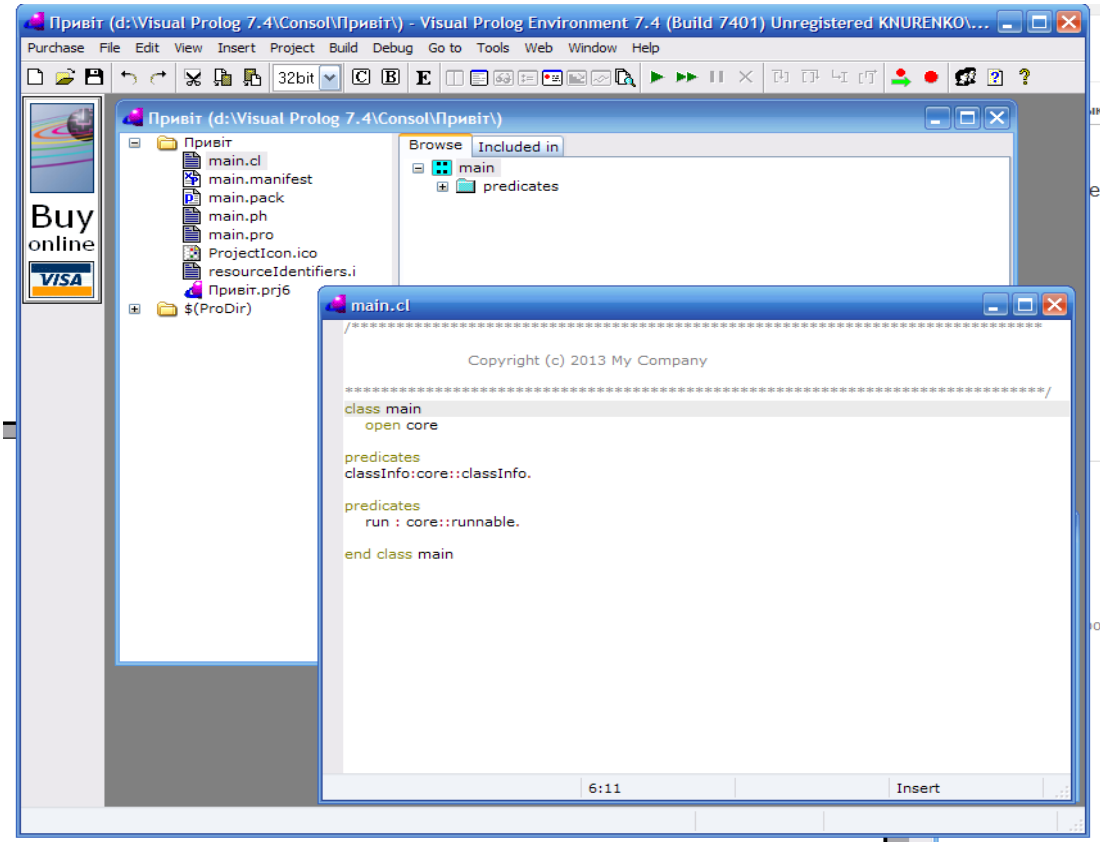


Рисунок 1.25. Вікно редактору коду, файл main.cl

```
class main
open core     % відкрити клас ядра Visual Prolog –  core
predicates
classInfo:core::classInfo.  % інформація про клас предикатів
Predicates
run : core::runnable.
end class main
```

Лістинг модуля реалізації класу main.pro наступний (рис. 1.26):

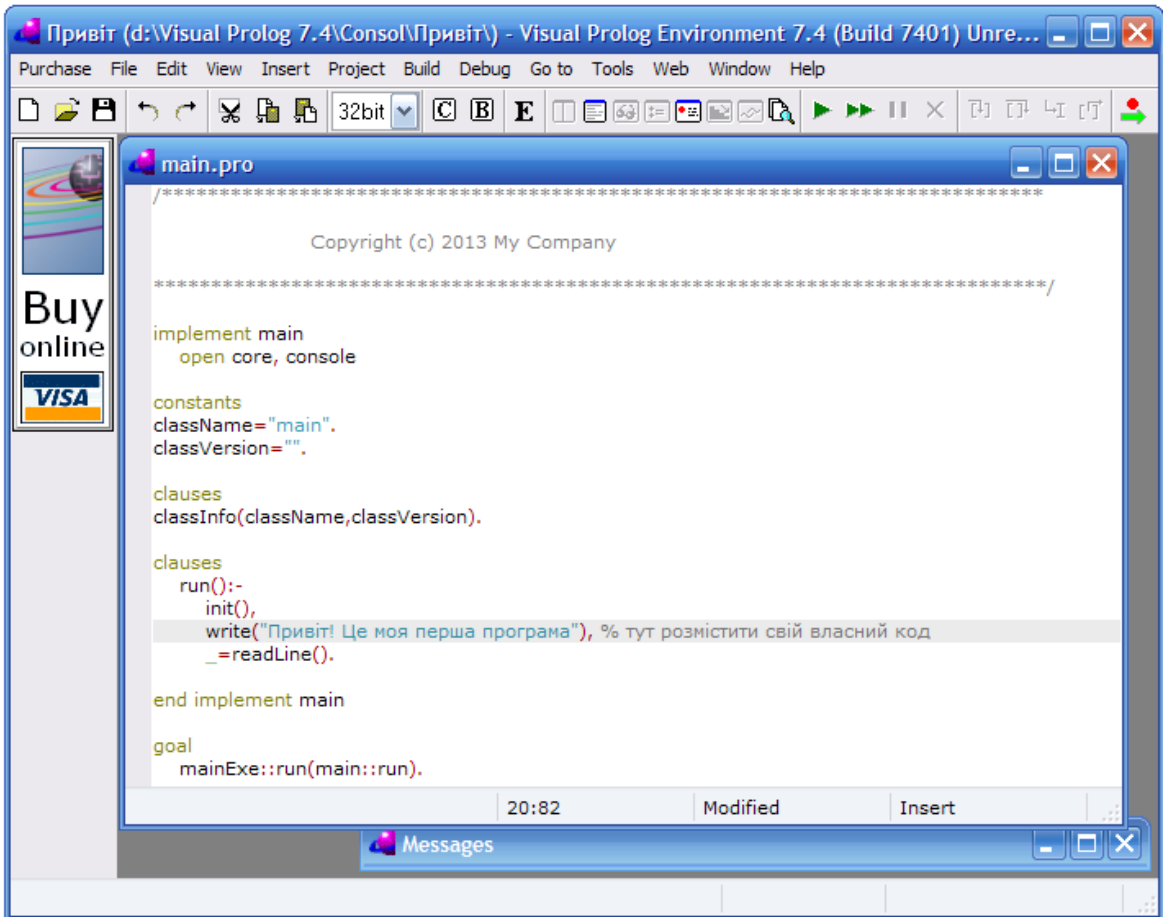


Рисунок 1.26. Вікно редактору коду, файл main.pro

```
implement main
        open core, console    % відкрити клас console
constants
className="main".
classVersion="".
Clauses
classInfo(className,classVersion).
Clauses
```

```
    run():-
    init(),        % ініціалізація консолі
    write("Привіт! Це моя перша програма!"), % виведення тексту
        _=readLine().    % очікування [Enter]
end implement main
goal
    mainExe::run(main::run).
```

При побудові проекту модуль реалізації формується за шаблоном, створюючи тим самим розділи:

- *implement main* – це заголовок, тип – реалізація (implement), модуль – головний (main);
- *constants* – це константи. Шаблон містить ім'я класу та його версію. Є можливість додавати власні константи;
- *Clauses* – це фрази. Шаблон містить інформацію про клас та його версію. Є можливість додавання власних фраз;
- *Clauses* – фрази, розділ у який програміст поміщає власний код;
- *end implement main* – оголошує кінець модулю;
- *goal* – ціль, у цьому розділі відбувається завантаження файлу *mainExe - mainExe::run(main::run)*.

1.2.2. Простір імен та реалізація

Простір імен – деяка множина, під якою мається на увазі модель, абстрактне сховище або оточення, створене для логічної угруповання унікальних ідентифікаторів (тобто імен).

Ідентифікатор, визначений у просторі імен, асоціюється з цим простором. Один і той же ідентифікатор може бути незалежно визначений у декількох просторах. Таким чином, значення, пов'язане з ідентифікатором, визначеним в одному просторі імен, може мати (або не мати) таке ж значення, як і такий же ідентифікатор, визначений в іншому просторі. Visual Prolog виконує підтримку просторів імен та визначає правила, що вказують, до якого простору імен належить ідентифікатор (тобто його визначення).

Простори імен представляють собою організації різних типів (доменів), які присутні у програмах Visual Prolog. Їх можна порівняти з текою у комп'ютерній файловій системі. Подібно текам, простори імен визначають для класів унікальні повні імена. Програма Visual Prolog містить одне або декілька просторів імен, кожне з яких або визначене програмістом, або визначене як частина написаної раніше бібліотеки класів.

Наприклад, простір імен PFC містить клас *concole*, який включає методи для читання та запису у вікні консолі.

При написанні класу поза оголошенням простору імен, компілятор надасть йому простір імен за замовчуванням.

Для використання методу *wrire*, визначеного у класі *concole* без попереднього визначення цього класу, слід використовувати рядок коду

Console :: write("Привіт! Це моя перша програма!").

Для спрощення програмування на початок вхідного файлу Visual Prolog доцільно вставити посилання на клас, що задає простір імен. Після цього можна використовувати код *write("Привіт! Це моя перша програма!")*.

Формат реалізації:

Implement ім'я_класу

ScopeQualifications(посилання на класи)
Розділи

end Implement

ScopeQualifications – це сфера кваліфікації, яка повинна містити:

- *open* –відкрити;
- *support* – список інтерфейсів, що підтримує даний клас;
- *Inherit* – спадковість;
- *Delegate* – делегування функціональності предикатів інтерфейсу з предикатів БД.

Розділи повинні містити:

- *Constants* – константи;
- *domains* – домени;

- *predicates* – предикати;
- *properties* – властивості;
- *Facts* – факти;
- *Clauses* – фрази;
- *Conditional* – умови.

1.2.3. Консольний додаток Привіт

Для створення слід вибрати команду меню **Project | New**. Далі заповнити діалогове вікно *Project Settings* та визначити (рис. 1.27):

- Ім'я проекту – Project Name;
- Тип проекту – Project Kind (GUI application чи Consol application);
- Базовий каталог – Base Directory. Вибір виконується за допомогою браузера, який відображає діалогове вікно для вибору теки у файловій системі;
- Підкаталог – Sub Directory, задається автоматично.

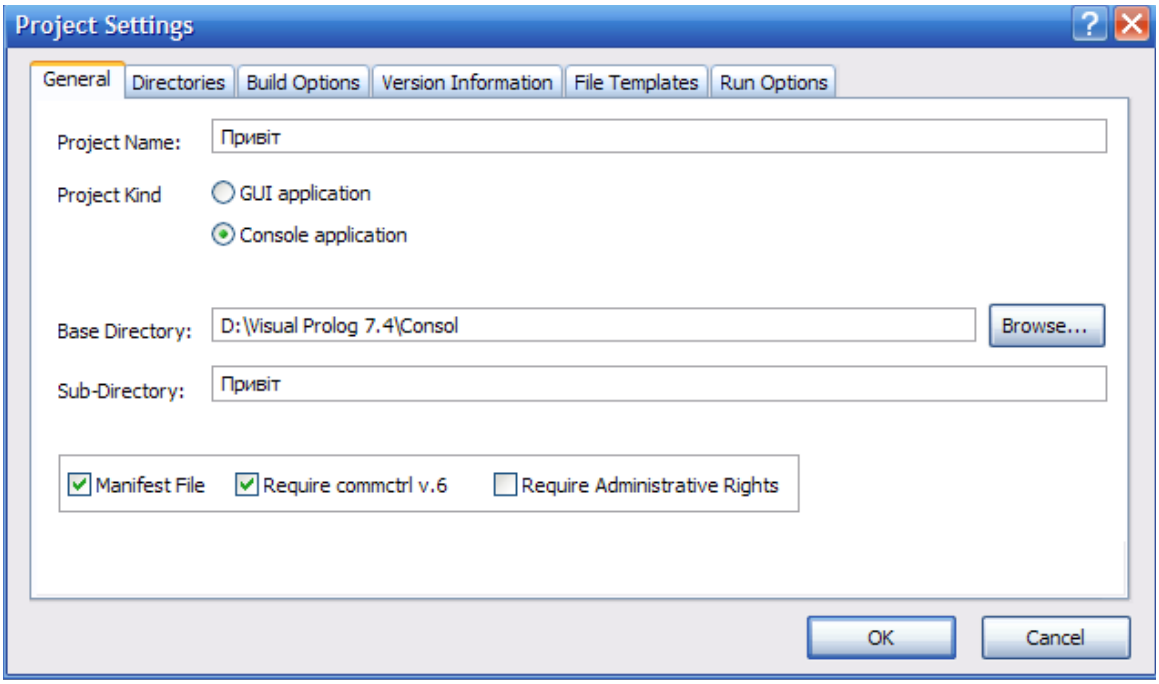


Рисунок 1.27. Вікно створення проекту

Для обраного проекту будується дерево (рис. 1.28), яке містить системні шаблони та вбудовані бібліотеки: зліва – дерево проекту, праворуч – браузер проекту з вкладкою *General*.

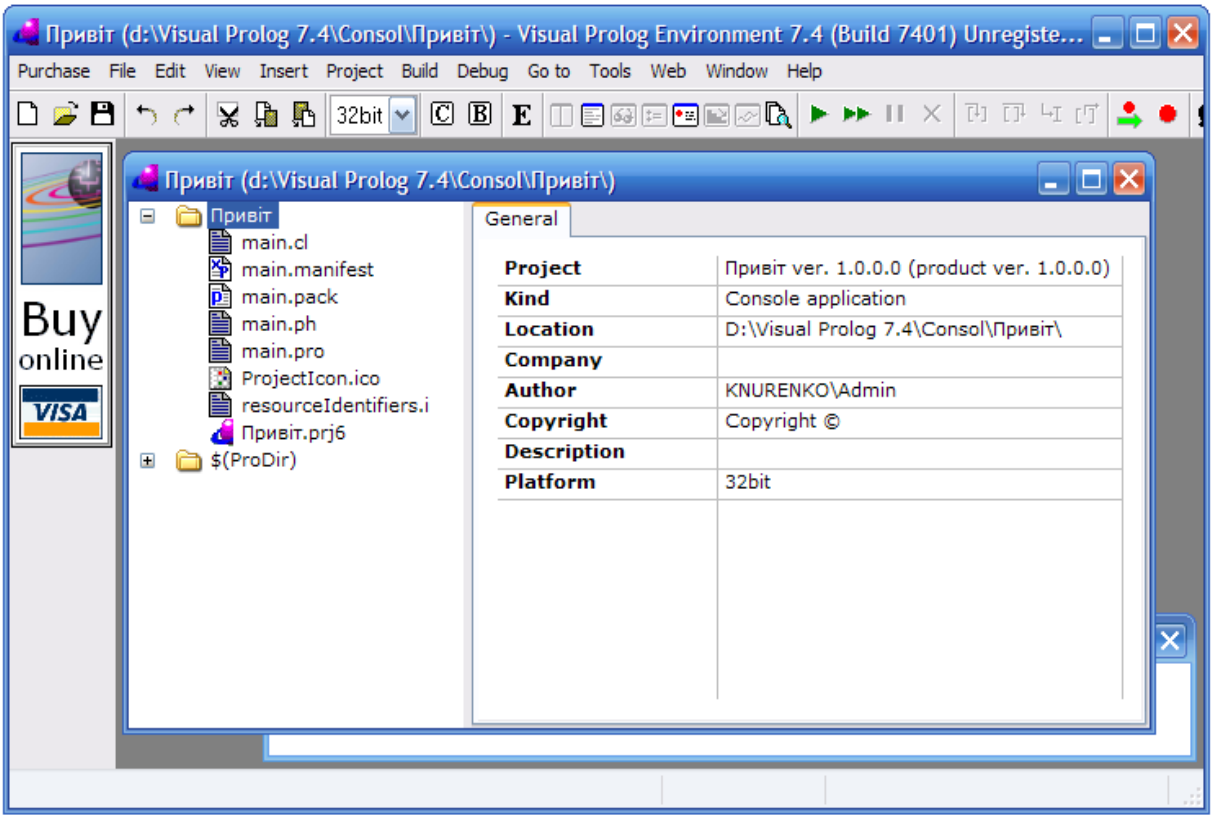


Рисунок 1.28. Стартове дерево проекту «Привіт»

Стартове дерево проекту «*Привіт*» включає системні компоненти, які не треба редагувати (інтегроване середовище розробки використовує їх при компіляції):

- Проект – Привіт.prj6;
- Маніфест для додатку – main.mainfest;
- Пакет для додатку – main.pack;
- Під'єднані системи бібліотек – $(ProDir)\lib.

У дерево проекту додаються файли проекту без функціональності (рис. 1.29):

- Шаблон класу – main.cl;
- Зміст проекту – main.ph;
- Шаблон реалізації додатку – main.pro.

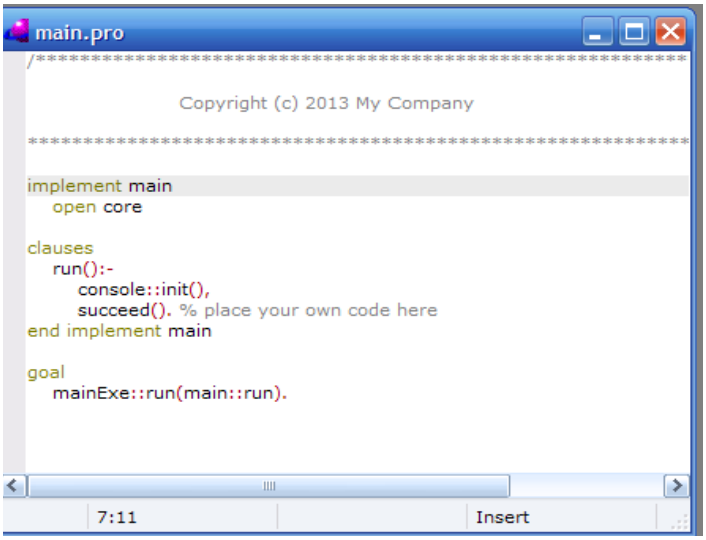


Рисунок 1.29. Стартовий код файлу main.pro проекту «Привіт»

Для додавання функціональності, обираємо файл *main.pro*, у шаблон якого слід додати власні операції. Діалогове вікно редактора коду відображає шаблон з коментарем, куди слід додати власний код (рис. 1.30).

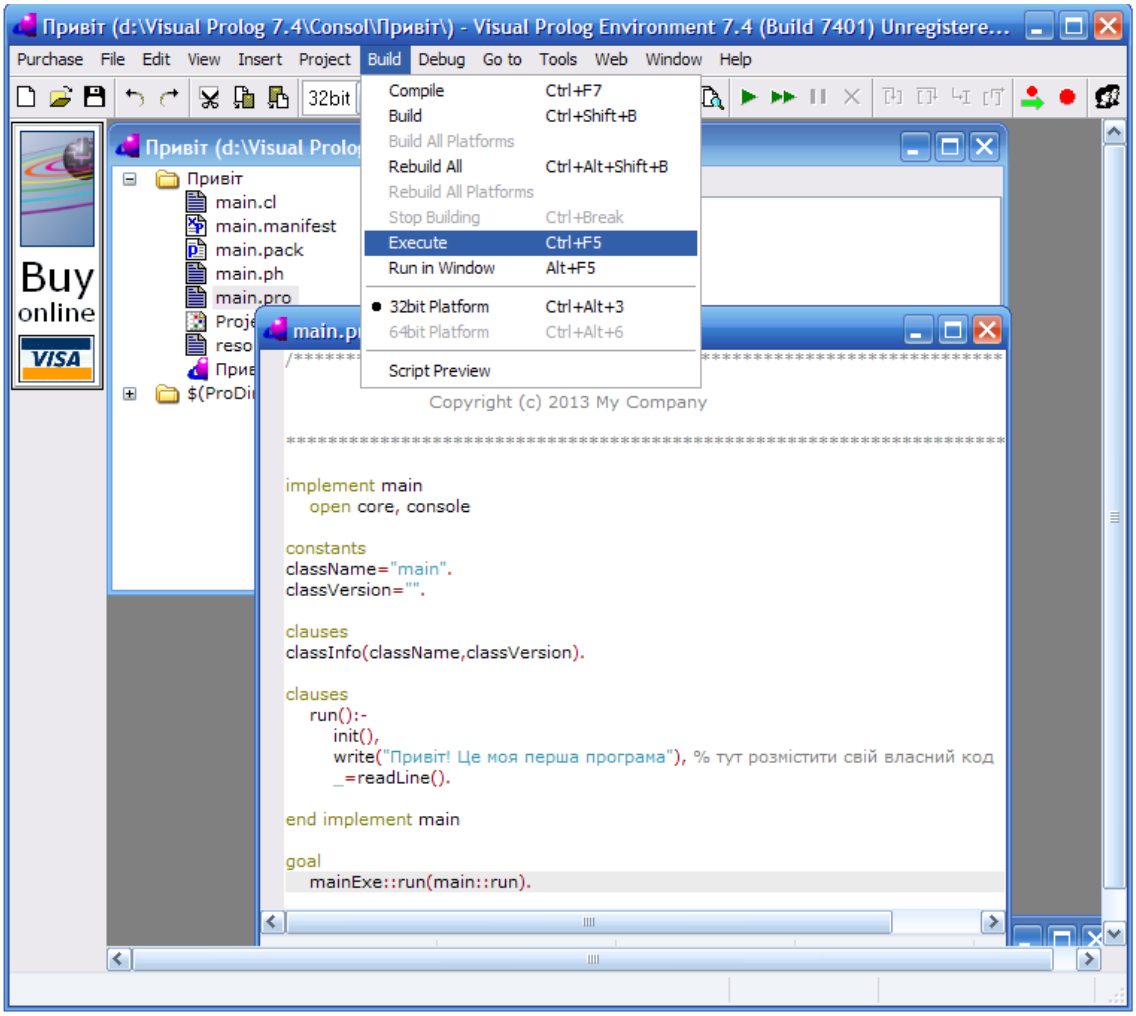


Рисунок 1.30. Виклик команди виконання проекту «Привіт»

У якості функціональності слід додати команду зчитування тексту введення *readLine()*.

А у фрагменті заголовку відкриваємо клас консолі (*console*), що дозволяє не згадувати консоль у предикаті *run()*, як наведено у лістингу:

```
implement main
open core, console    % відкриваємо клас console

constants
className="main".
classVersion="".

Clauses
classInfo(className,classVersion).
Clauses
run():-
init(),          % ініціалізація консолі
write("Привіт! Це моя перша програма!"),   % виведення тексту
_=readLine().          % очікування [Enter]
end implement main

goal
mainExe::run(main::run).
```

Під час завантаження проекту командою меню **Build | Execute** (рис. 1.30) з'являється вікно Activate (рис. 1.31) з пропозицією активізувати (придбати коштовну версію).

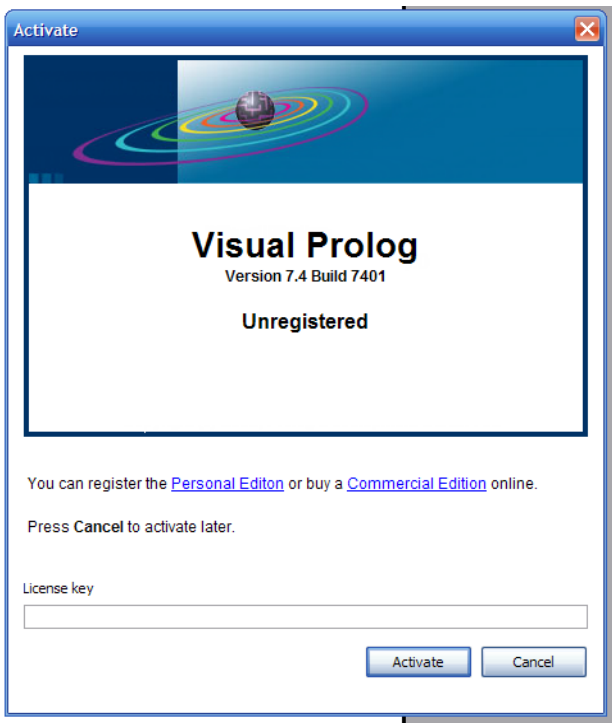


Рисунок 1.31. Пропозиція Activate для придбання коштовної версії

Слід натиснути кнопку *Cancel* і на екрані з'явиться повідомлення (рис. 1.32) про те, що цей додаток створено у версії Visual Prolog Personal Edition і що не можна розповсюджувати або використовувати його в комерційних цілях.

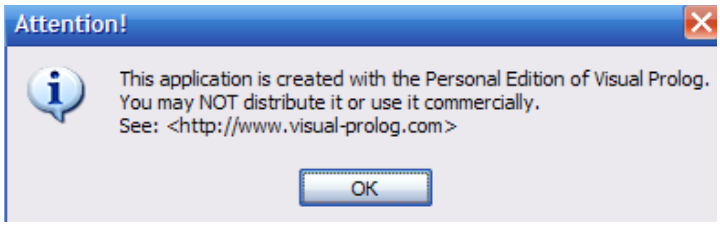


Рисунок 1.32. Попереджувальне повідомлення

Після погодження повідомлення Пролог виводить результат у вікні, як показано на рис. 1.33, з текстом «*Привіт! Це моя перша програма!*». Для завершення слід натиснути клавішу [*Enter*].

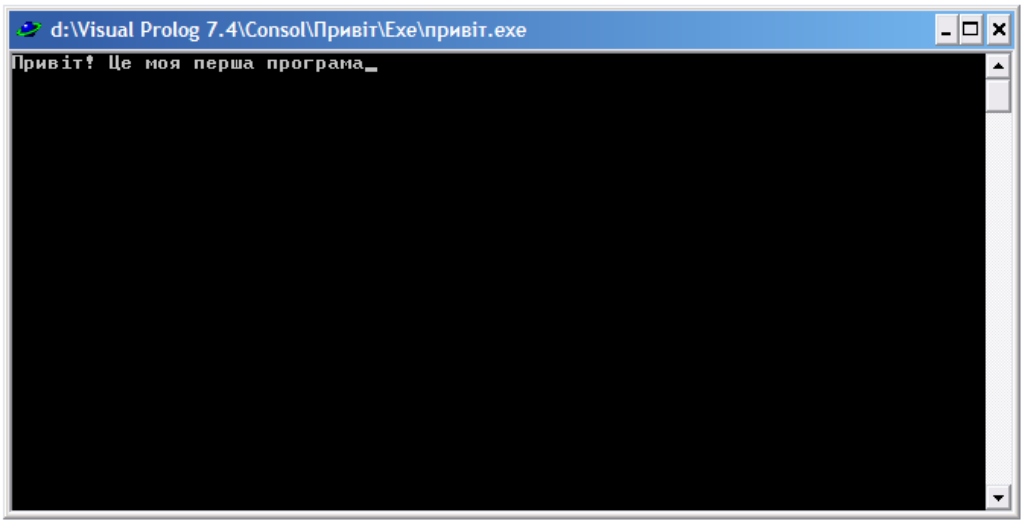


Рисунок 1.33. Вікно виведення результату виконання

Після компіляції автоматично відображається вікно повідомлень *Massages* з інформацією про збереження компонентів проекту та створення пакету *main.pack* (рис. 1.34).

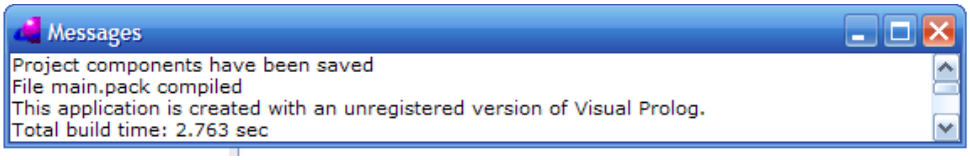


Рисунок 1.34. Вікно повідомлень

При наявності помилок коду виводиться вікно повідомлення (рис. 1.35), яке містить інформацію про номер (код) помилки, про рядок та символ у рядку.

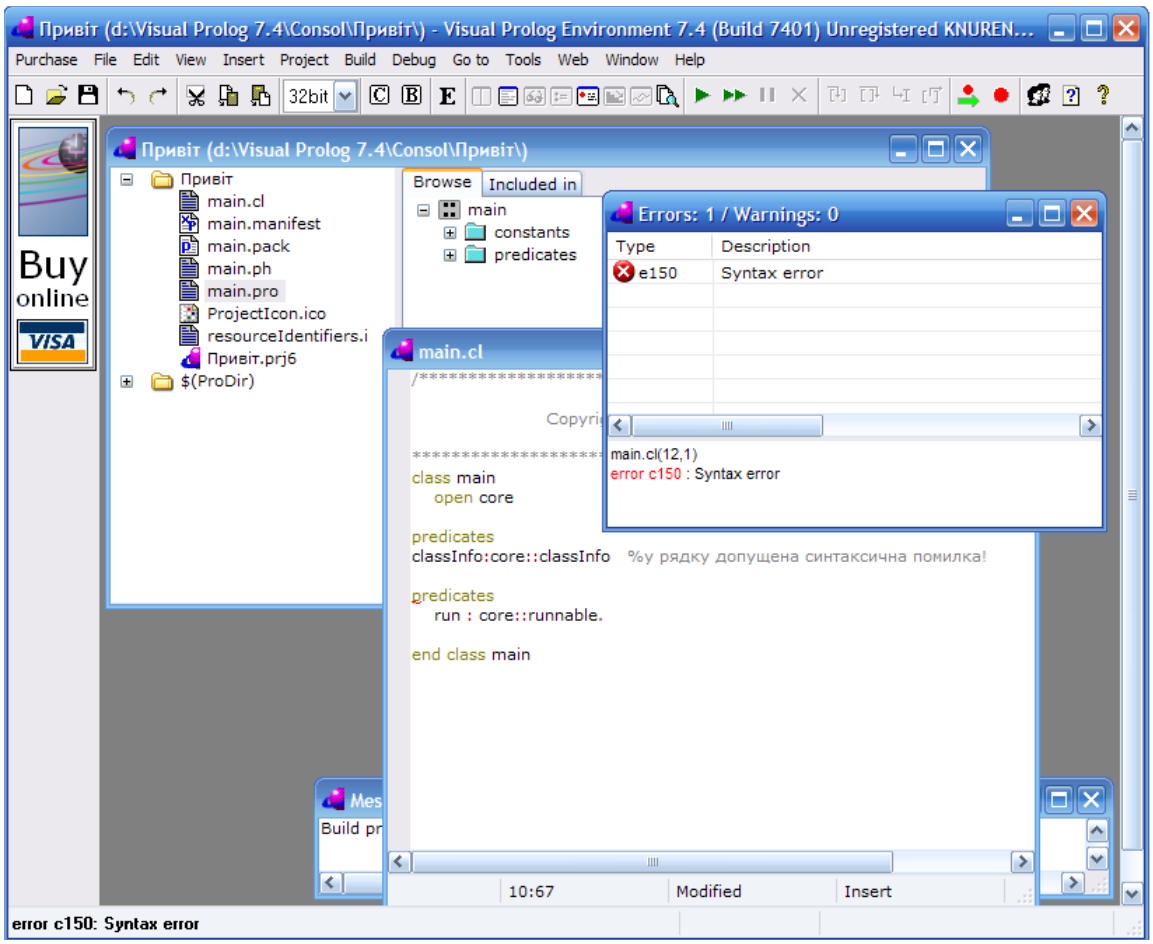


Рисунок 1.35. Повідомлення про помилки

Відповідно у лістингу виконуються помітки помилок підкресленням червоним кольором. Недоліком є той факт, що лістинг не містить нумерації рядків.

Інтегроване середовище розробки містить інтелектуальний покажчик, який у момент набору символу коду відображається список можливих продовжень для набору. Таким чином позбавляє програміста від запам'ятовування та ручного набору попередньо визначених кодів. Якщо продовження коду має варіанти, то вони перечислюються праворуч у діалоговому вікні (рис. 1.36).

Для правильних кодів використовується кольорова семантична підсвітка. Наприклад, рядки відображаються блакитним кольором.

Підкажчику доступні ідентифікатори, що містяться у класах проекту. За замовчуванням обирається тільки клас ядра *core*.

Для роботи з консоллю слід додати клас *console*, а для роботи з математичними функціями – клас *math*. Якщо цього не зробити, то у коді перед

ідентифікатором елемента класу потрібно додавати ім'я класу та розділювач (вертикальна двокрапка).

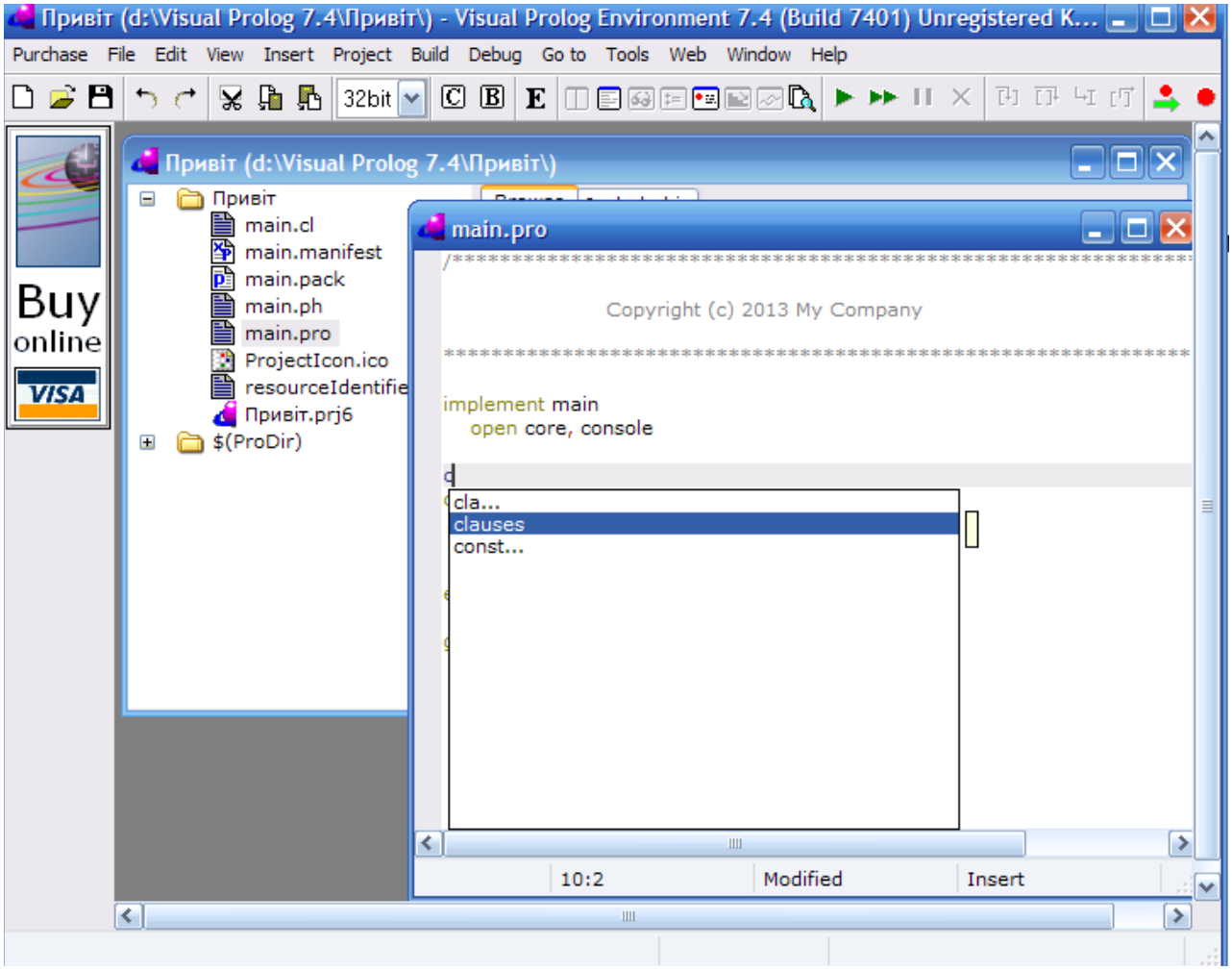


Рисунок 1.36. Приклад роботи інтелектуального покажчика

1.2.4. Створення проекту GUI у Visual Prolog

Завантаження Visual Prolog відкриває IDE системи, меню якого можна визначити як меню задач (TaskMenu).

Система вікон і діалогів, що створюється для спілкування з користувачами програми, називається Graphical User Interface (GUI).

Для створення проекту у діалоговому вікні, що викликається командою меню задач **Project | New** (рис. 1.37), слід виконати:

- привласнити ім'я проекту (Project Name);
- визначитися з текою (виконується за допомогою браузера Browse, який викликає діалогове вікно файлів. Поточна тека створюється автоматично);

- обрати тип проекту GUI Application (Project Kind).

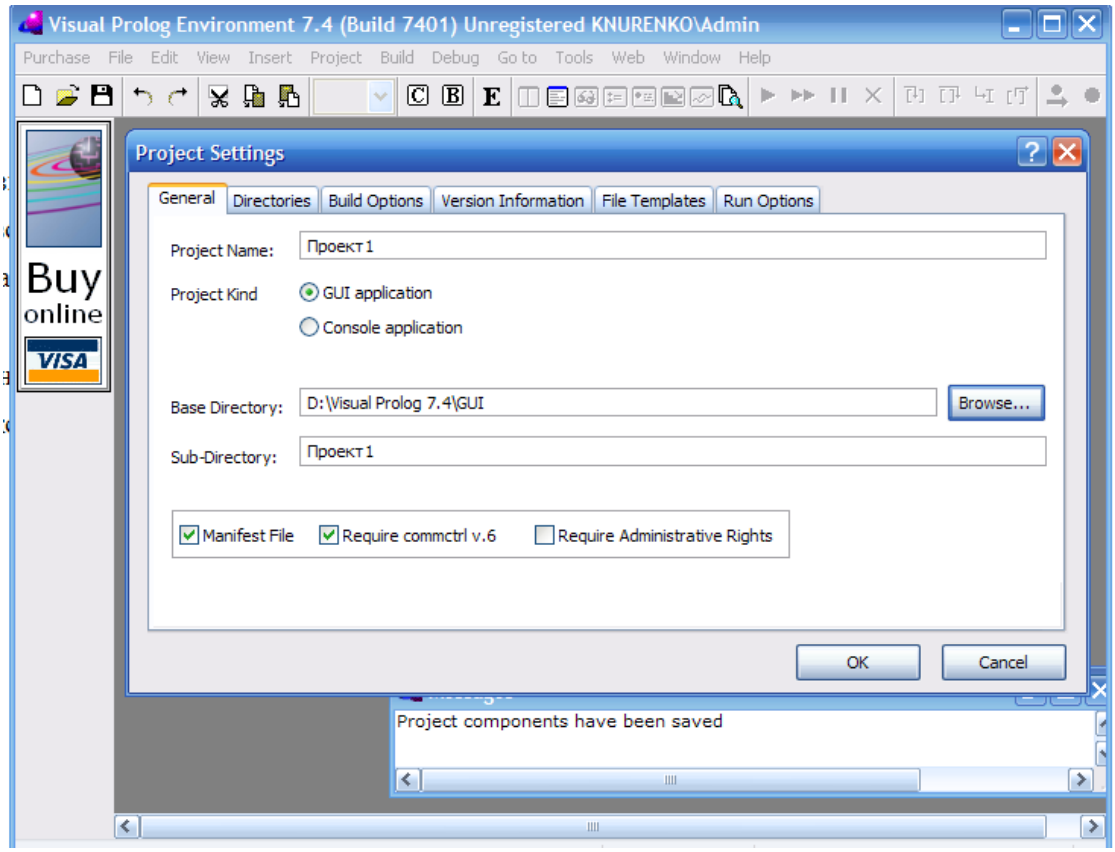


Рисунок 1.37. Вікно створення проекту

Після натиснення кнопки *OK*, з'являється вікно шаблону дерева проекту (рис. 1.38).

Найвищий рівень дерева проекту представляє сам проект і проектна директорія.

Наступний нижчий рівень – логічний вузол *$(ProDir)*, який представляє директорію, де встановлена сама система Visual Prolog. Ця директорія містить бібліотеки (*libraries*) та бібліотеки вихідних текстів системи Visual Prolog.

Директорія *TaskWindow* є піддиректорією проектної директорії та містить всі необхідні коди для створення головного вікна додатку (*Task Window*), його меню (*Menu*), панель інструментів (*Toolbar*) та діалог інформації (*about dialog*).

Visual Prolog використовує наступні угоди, про що свідчить наявність файлів:

- *.ph* файли є заголовками пакетів (*package headers*). Пакет є набором класів и інтерфейсів, які спільно використовуються;

- *pack* файли є *packages*. Вони містять виконуючі розділи або конкретизації файлів, що перераховані у відповідних *.ph* файлах;
- *.i* файл містить інтерфейс (*interface*);
- *.cl* файл містить декларацію класу (*class declaration*);
- *.pro* файл містить імплементацію класу (*class implementation*).

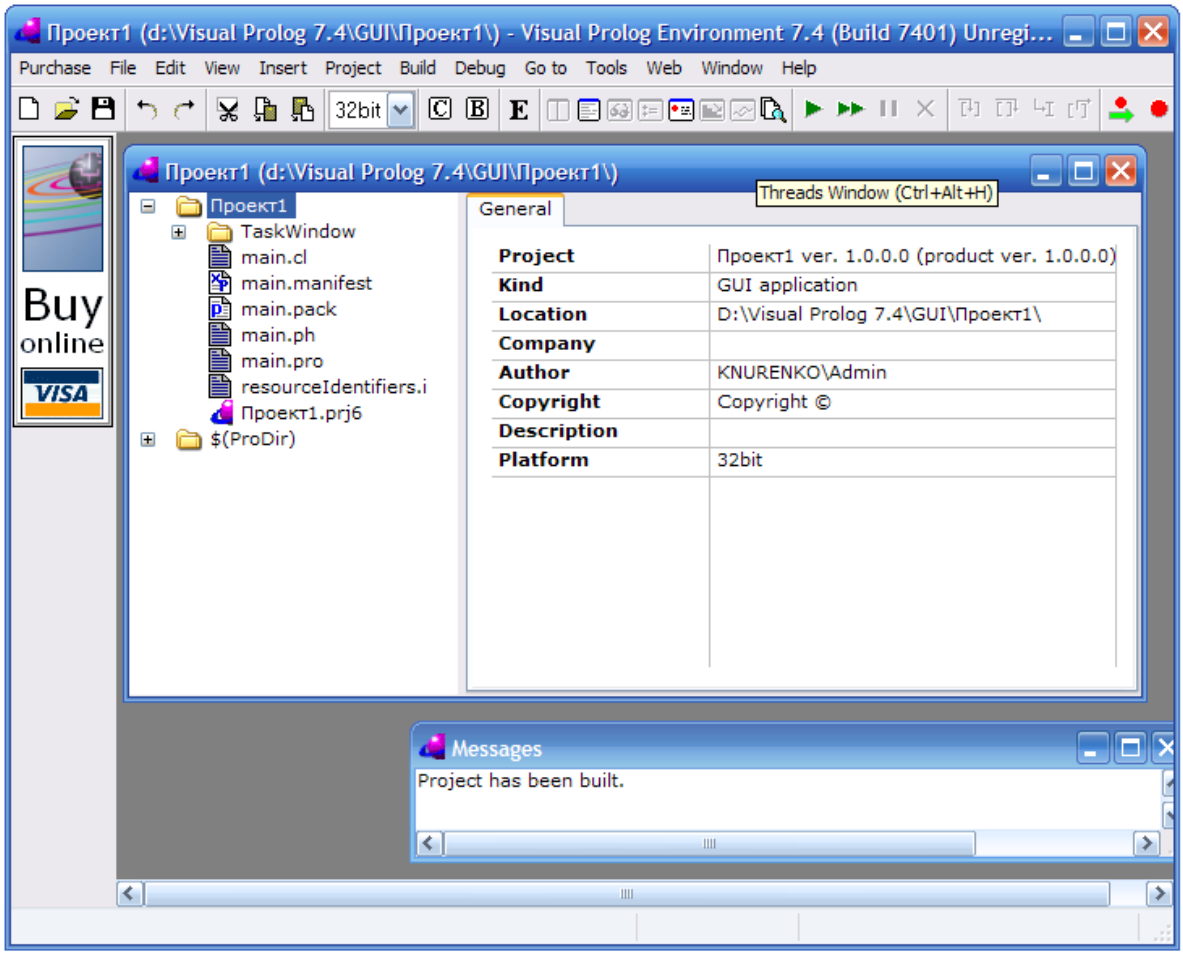


Рисунок 1.38. Дерево проекту

Для завантаження програми слід виконати команду **Build | Execute**. При цьому з’явиться вікно з пропозицією активізувати Visual Prolog. У відповідь натиснути кнопку *Cancel*, після чого з’явиться наступне вікно (рис. 1.39).

Для виходу з програми слід скористатися кнопкою управління вікном або командою меню додатку **File | Exit**.

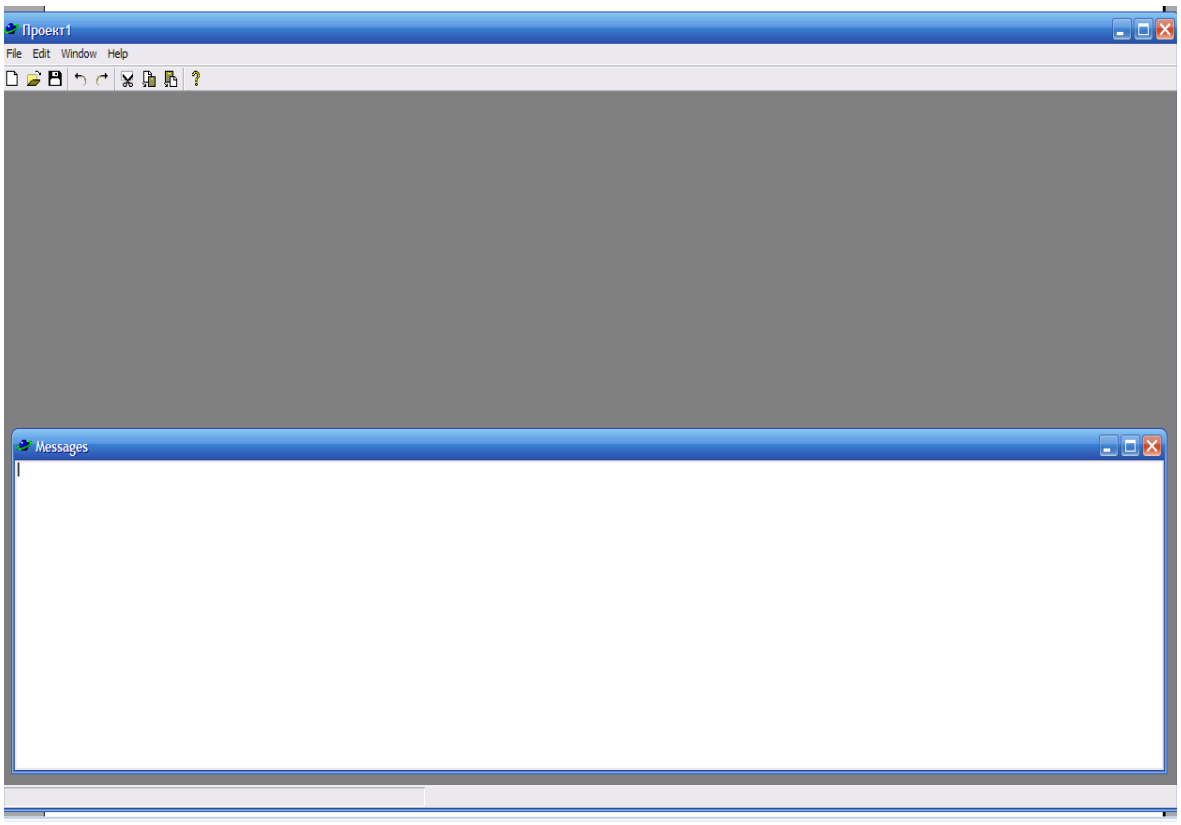


Рисунок 1.39. Вікно проекту

1.2.5. Компіляція та завантаження програми

Механізм створення проекту виконується шляхом створення ресурсів, постійних файлів, поновлення секцій вихідного коду програмістами, компіляції і компонування всіх модулів.

Виконується перевірка дати створення файлів, щоб визначити, які файли повинні бути скомпільовані або які перекомпілювати (якщо файл типу *.OBJ* має новішу версію або якщо вихідні файли включають будь-який файл, що не існує).

Створений об'єкт отримує необхідну інформацію з файлу проекту (*<ProjectName>. Prj6*), включає в себе набір вихідних файлів і створених скриптів.

Сценарії побудови складаються з наступних частин:

- основний сценарій побудови описує, як виконати кінцеву мету;
- правила описують як компілювати файли з заданими розширеннями;

- символи – це символічні імена, які можуть бути використані у створенні сценарію і в побудові правил.

При виконанні проекту параметри сценарію генеруються за замовчуванням, які потім можуть бути змінені і розширені програмістом.

Налаштування для виконання проекту повинні бути вказані в налаштуваннях проекту (**Project | Settings**), нарешті, створений скрипт можна переглянути у вікні перегляду діалогу цього сценарію (**Build | Script Preview**).

Створені скрипти можуть послатися на символи. Синтаксис, який використовується в сценарії для позначення певного символу – *$ (<symbol>)*. Ці символи можуть бути:

- визначені символи, що автоматично обробляються IDE, наприклад, *project_obj*, *project_lib*, *, ** і т. д. (які не можуть бути безпосередньо змінені програмістом).
- початкові символи, що визначаються IDE (породжені Project Settings), але які потім можуть бути змінені програмістом, наприклад, *incdir*, *exedir*, *objdir*. Програміст може активувати параметри проекту, а потім змінити/встановити ці символи з вкладки Каталоги (команда **Project | Settings**).

Використовуються деякі стандартні правила при компіляції файлів. Встановлено наступний синтаксис правила:

InputExtension -> OutputExtension: CommandLine

Розширення введення -> Розширення виведення: Командний рядок

Кожен рядок є командою *CommandLine*, якою потрібно конвертувати файл з одним розширенням *InputExtension* у файл з іншим розширенням *OutputExtension*.

Команда **Build | Compile** намагається виконати компіляцію модуля, що містить обраний файл. IDE не зможе компілювати файл, який не є частиною відкритого проекту.

Команда **Build | Build** перевірить дату проекту. Якщо всі ресурси були змінені з моменту створення, код експертів можливо оновить деякі розділи у

вихідному файлі перед побудовою. Ця команда буде будувати проект, перевіряючи тимчасові мітки всіх вихідних файлів у проекті. Тому, якщо будуть версії вихідних файлів більш нові, ніж попередніх *.OBJ* файлів, то відповідні модулі проекту буде перекомпільовано.

Побудова проекту також буде спиратися на файли ресурсів. Після цього проект буде пов'язаний зі створеним цільовим модулем (виконуваних програм або *DLL*).

При необхідності команда **Build | Execute** буде робити перші побудови (**Build | Build**), а потім буде завантажений згенерований виконуваний файл.

Build | Execute. Команда *Run* (виконати) доступна тільки у вікні для консольних додатків. *Run* намагається виконати цільовий модуль поточного проекту у спеціально створеному вікні в текстовому режимі. При необхідності, ця команда буде робити перші **Build | Build** (генерувати виконуваний файл), а потім створює і відображає вікно у текстовому режимі. У цьому вікні у режимі DOS на питання програмісту: «Натисніть будь-яку клавішу для продовження ...» – після натиснення згенерований виконуваний файл буде завантажений. Завжди можна зупинити налаштування сесії командою **Debug | Stop Debugging**.

Коли розпочинається налаштування (відладка), IDE перш за все завантажує інформацію для відладки і тільки потім приступає до запуску програми. Дія IDE призупиняється безпосередньо перед початком виконання розділу *goal*. Для цього відкривається текстовий редактор і покажчик встановлюється на ключове слово *goal* (рис. 1.40).

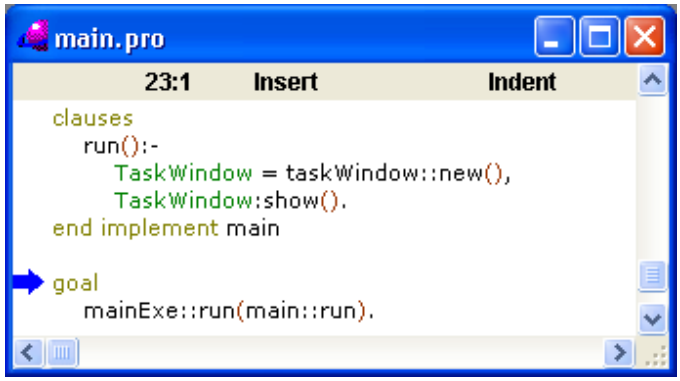


Рисунок 1.40. Виконання goal

1.2.6. Створення теми проекту

Установка 32-бітної чи 64-бітної платформи проекту (такі підтримуються ОС: WinXP, Windows Vista, Windows Vista x64, Windows 7, Windows 7 x64), є як керівництво для побудови, виконання відповідного проекту.

Для виконання конкретних дій у проекті слід створити нову тему, скориставшись командою меню **File**:

- *New in New Package*… – створити нову тему у новому пакеті (рис. 1.41);
- *New in Existing Package* … – створити нову тему в існуючому пакеті;

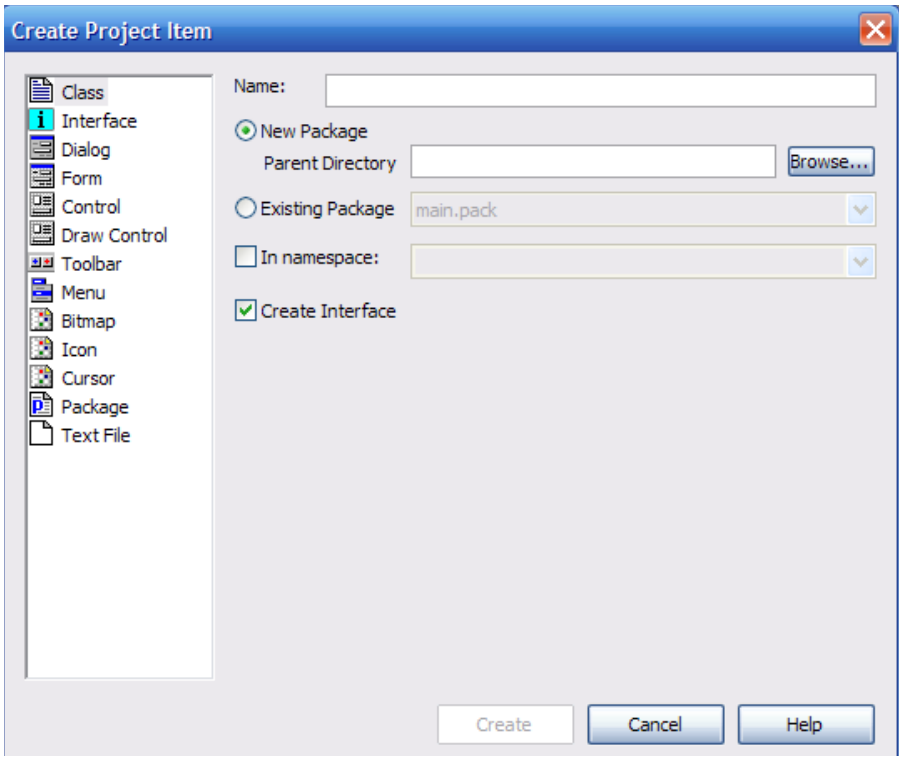


Рисунок 1.41. Виконання File | New in New Package…

Для генерації вікна створення теми *Greate Project Item* можна створювати наступні теми:

- Package – вбудований пакет;
- Class – клас;
- Interface – інтерфейс;
- Dialog – діалог;

- Form – форма;
- Control – елемент управління;
- Toolbar – панель інструментів;
- Menu – меню;
- Icon – іконка;
- Cursor – курсор;
- TextFile – текстовий файл.

Наприклад, оберемо форму (*Form*), як показано на рис. 1.42.

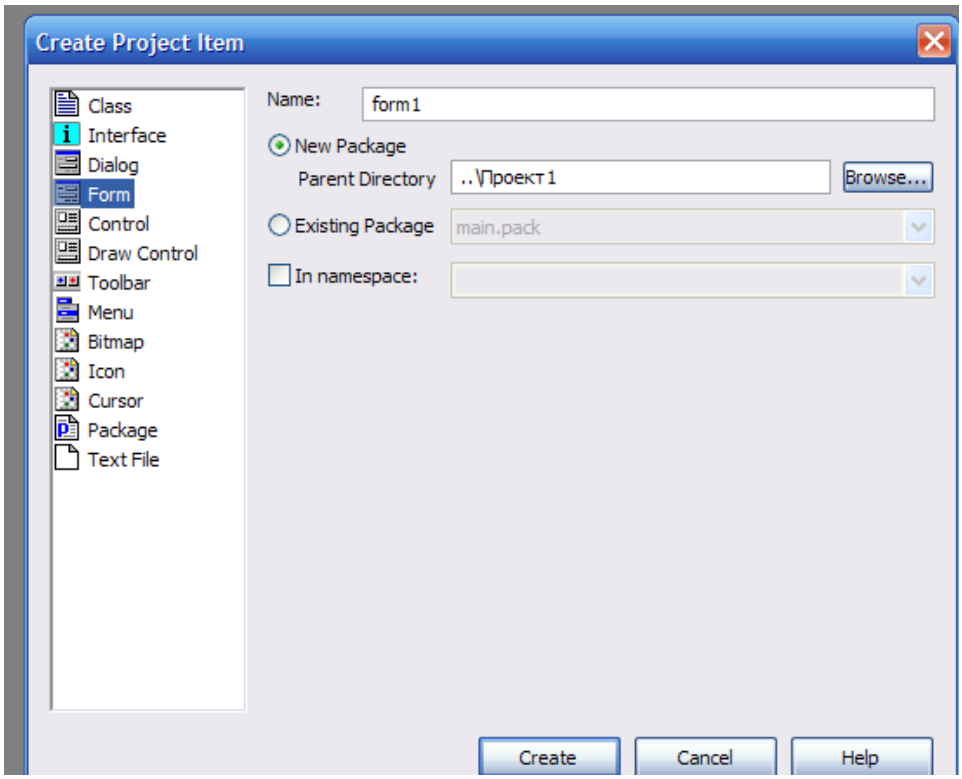


Рисунок 1.42. Створення форми

Після натиснення кнопки *Create*, генерується вікно (рис. 1.43) конструктора форми, що містить наступні вбудовані вікна:

- Form1 – форма;
- Properties – властивості;
- Controls – елементи управління, які можна використовувати у формі методом переміщення;
- Layout – засоби управління положенням компонентів у формі.

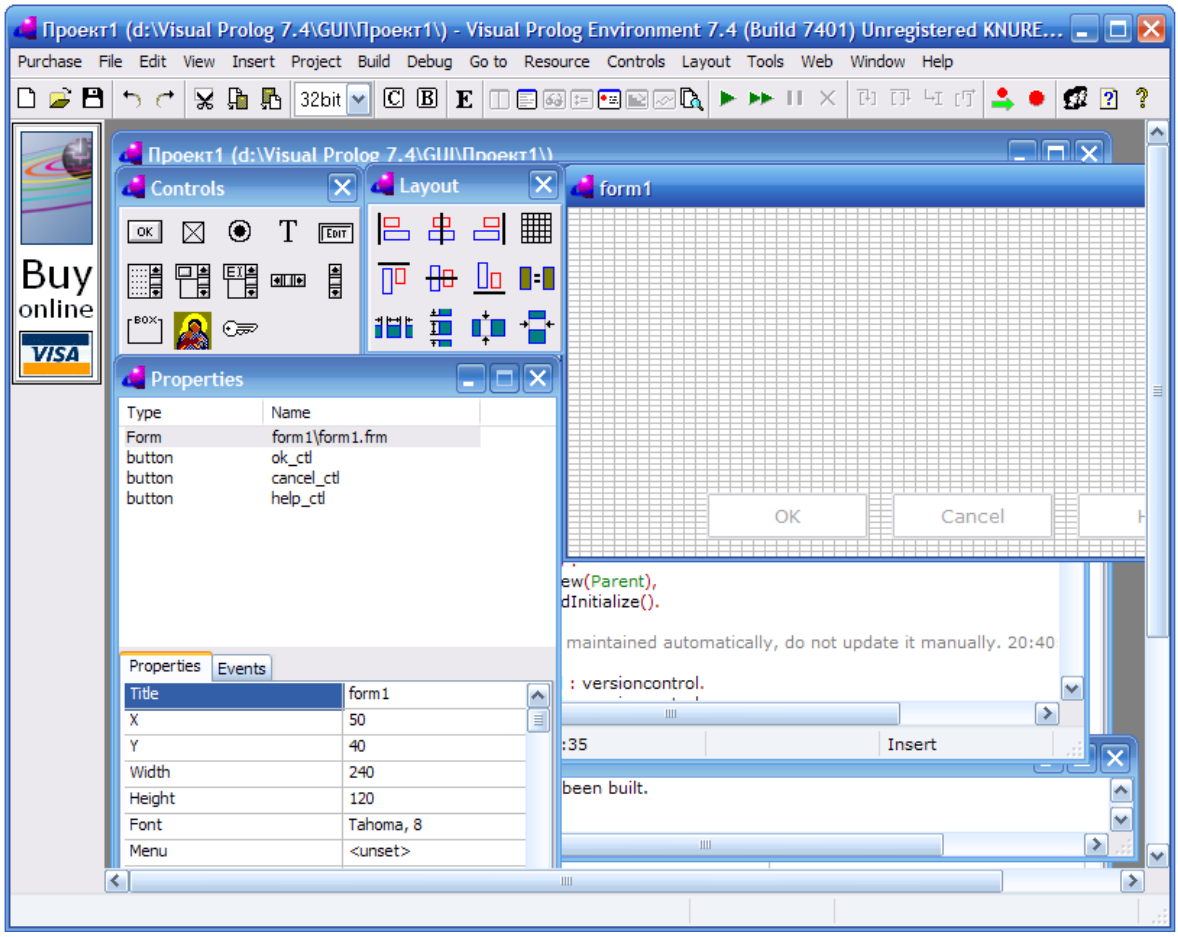


Рисунок 1.43. Конструктор форми

Для регулювання поведінки вікна проекту у дереві проекту треба визвати поле *TaskWindow.win* (рис. 1.44).

У цьому діалоговому вікні можна спостерігати декілька нових типів вузлів:

- *.dlg* файл містить діалог (dialog);
- *.frm* файл містить форму (form);
- *.win* файл містить вікно (window) (вікно додатку або звичайне вікно класу window з PFC GUI);
- *.mnu* файл містить меню (menu);
- *.ico* файл містить іконку (icon).

Також проект може містити:

- *.ctl* файли, що містять елементи управління (controls);
- *.tb* файли, що містять панелі інструментів (toolbars);
- *.cur* файли, що містять курсори (cursors);
- *.bmp* файлы, що містять картинки (bitmaps);

- *.lib* – це бібліотеки (libraries).

Права кнопка миші викликає контекстне меню з операціями, які можна виконати над поточним вузлом. Подвійне клацання на вузлі викликає редактор відповідного елемента. Всі коди редагуються за допомогою текстового редактора, а ресурси на основі вікон, такі як діалоги та меню редагуються за допомогою графічних редакторів.

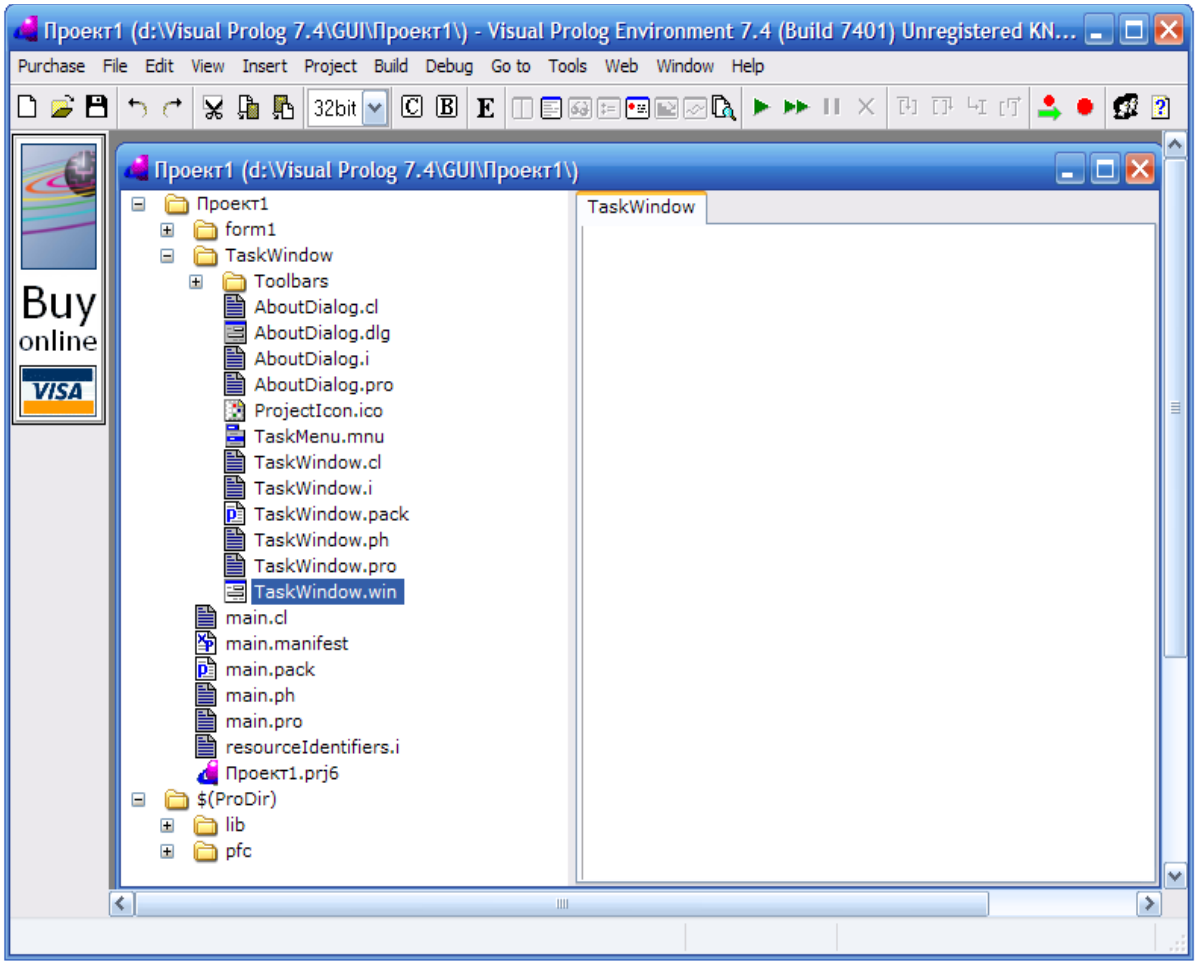


Рисунок 1.44 .Команда TaskWindow.win для form1

Файл *main.cl* містить клас main, який містить два предиката з іменами *classinfo* та *run*, що являються підпрограмами.

Деякі сутності представлені в дереві двічі, оскільки вони мають як декларацію, так і імплементацію. Наприклад (рис. 1.45), предикат *run* у класі *main*.

Подвійне клацання на кожному вузлі з іменем *run* буде відкривати два редактора, що показують декларацію та імплементацію предиката *run*, відповідно.

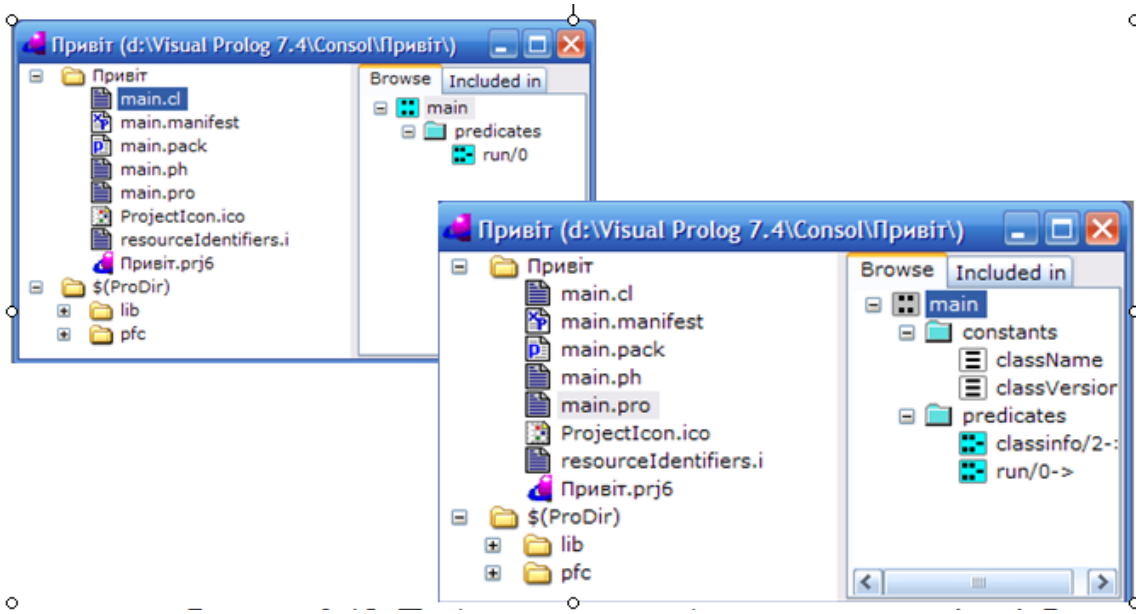


Рисунок 1.45. Представлення предикату run у класі main

Подвійний клік на *TaskWindow.win* (рис. 1.44) відкриває вікно (рис. 1.46) *Dialog and Window Expert* (*TaskWindow*). У відкритому вікні є можливість змінити деякі параметри, які встановлені за замовчуванням та не дозволяють деякі дії.

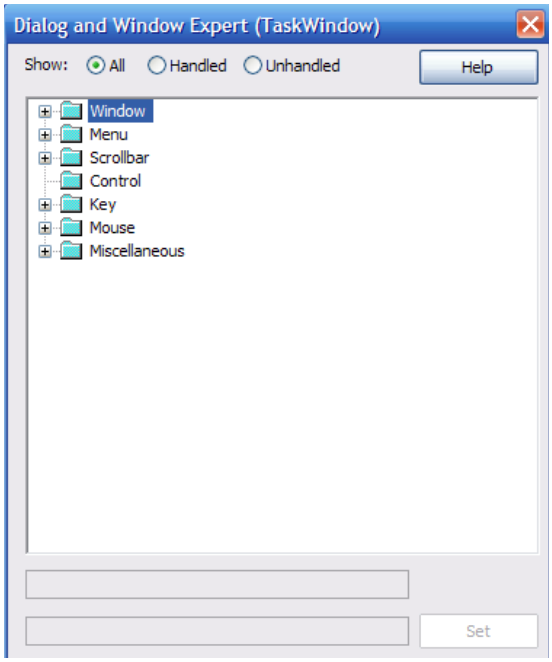


Рисунок 1.46. Вікно Dialog and Window Expert

Для поля вікон (Windows) треба зробити доступними (рис. 1.47):

- *Destroy* – знищення;
- *Show* – відображення;
- *Size* – розмір.

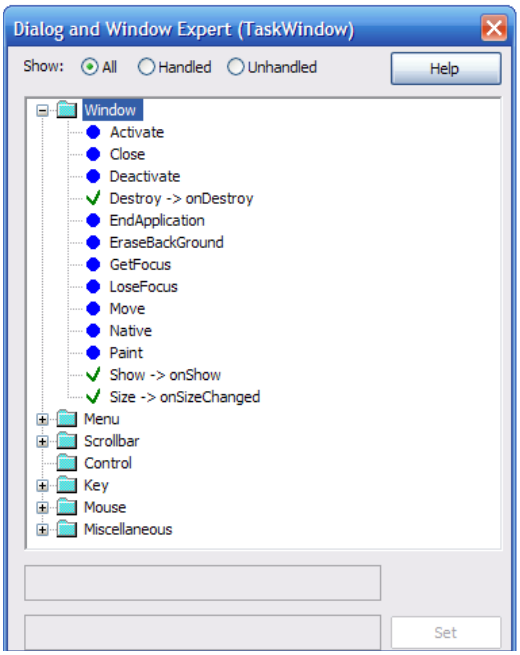


Рисунок 1.47. Параметри вікон

Для поля *Menu*, як показано на рис. 1.48, треба зробити доступними параметри. Для підключення пункту треба виконати на ньому подвійний клік мишею:

- *in_file_new* – відкрити новий файл;
- *in_file_exit* – закрити файл;
- *id_help_about* – викликати довідку.

Інші поля змінювати не рекомендується. Поля без прапорця вважаються не підключеними.

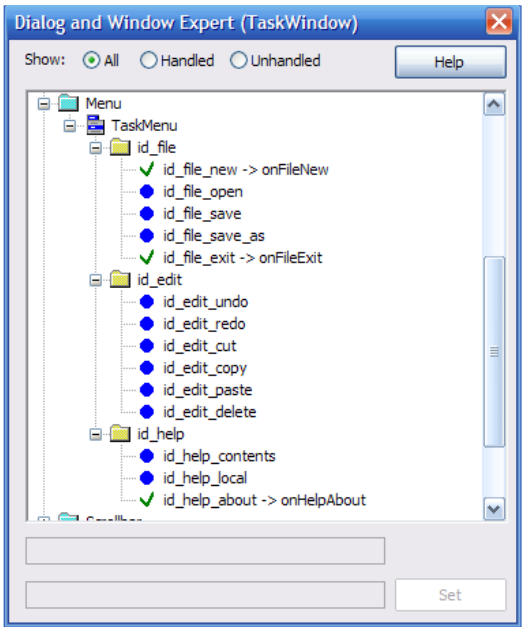


Рисунок 1.48. Параметри поля Menu

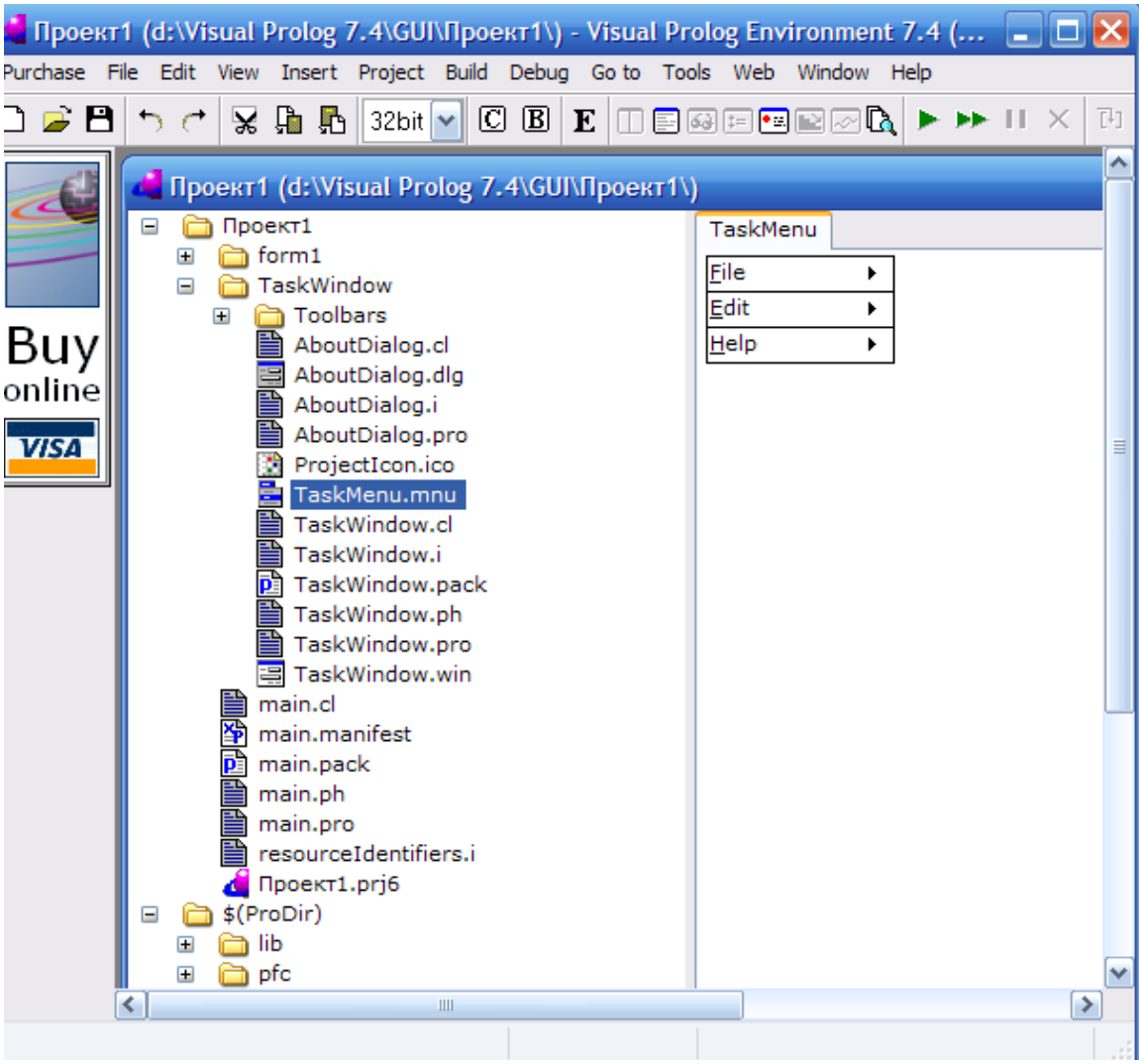


Рисунок 1.49. Відкриття TaskMenu

Для відкриття вікна форми, потрібно дозволити відкриття файлу форми (за замовчуванням це не дозволено). Файл *TaskMenu* відкривається подвійним кліком по *TaskMenu.mnu* (рис. 1.49).

Відкривається вікно *TaskMenu*, у якому для *&File\>&New\F7* слід зняти прапорець в *Disable* (рис. 1.50).

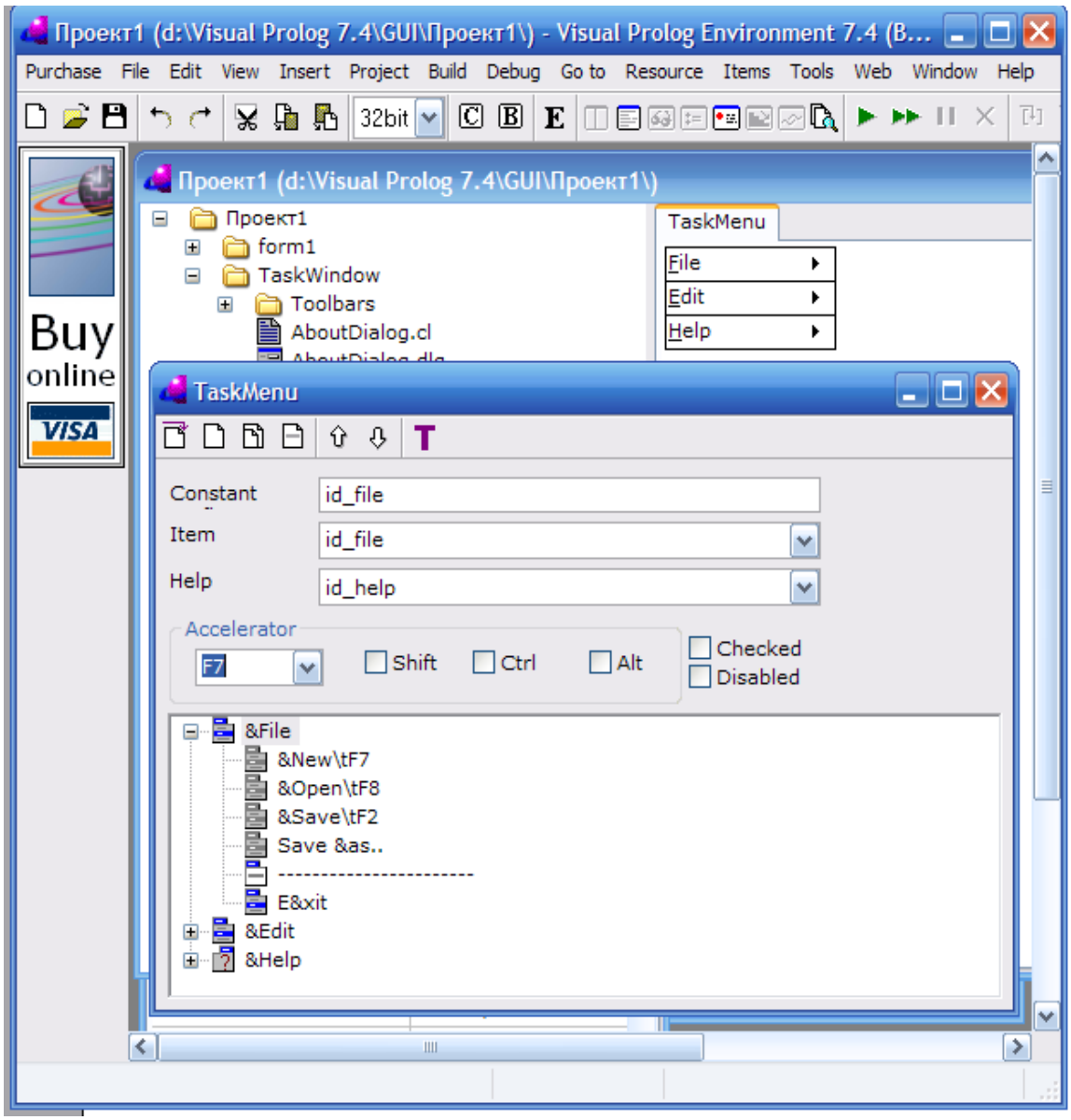


Рисунок 1.50. Параметри вікна TaskMenu

У момент закриття цього вікна з'являється запит (рис. 1.51) на збереження виконаних змін. Треба обрати *Save* (зберегти).

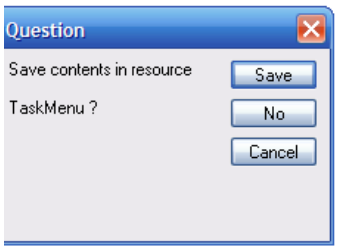


Рисунок 1.51. Запит

Залишається прописати посилання на файл, що відкривається. Слід відкрити файл *TaskWindow.pro* (рис. 1.52) подвійним кліком по *TaskWindow*.pro.

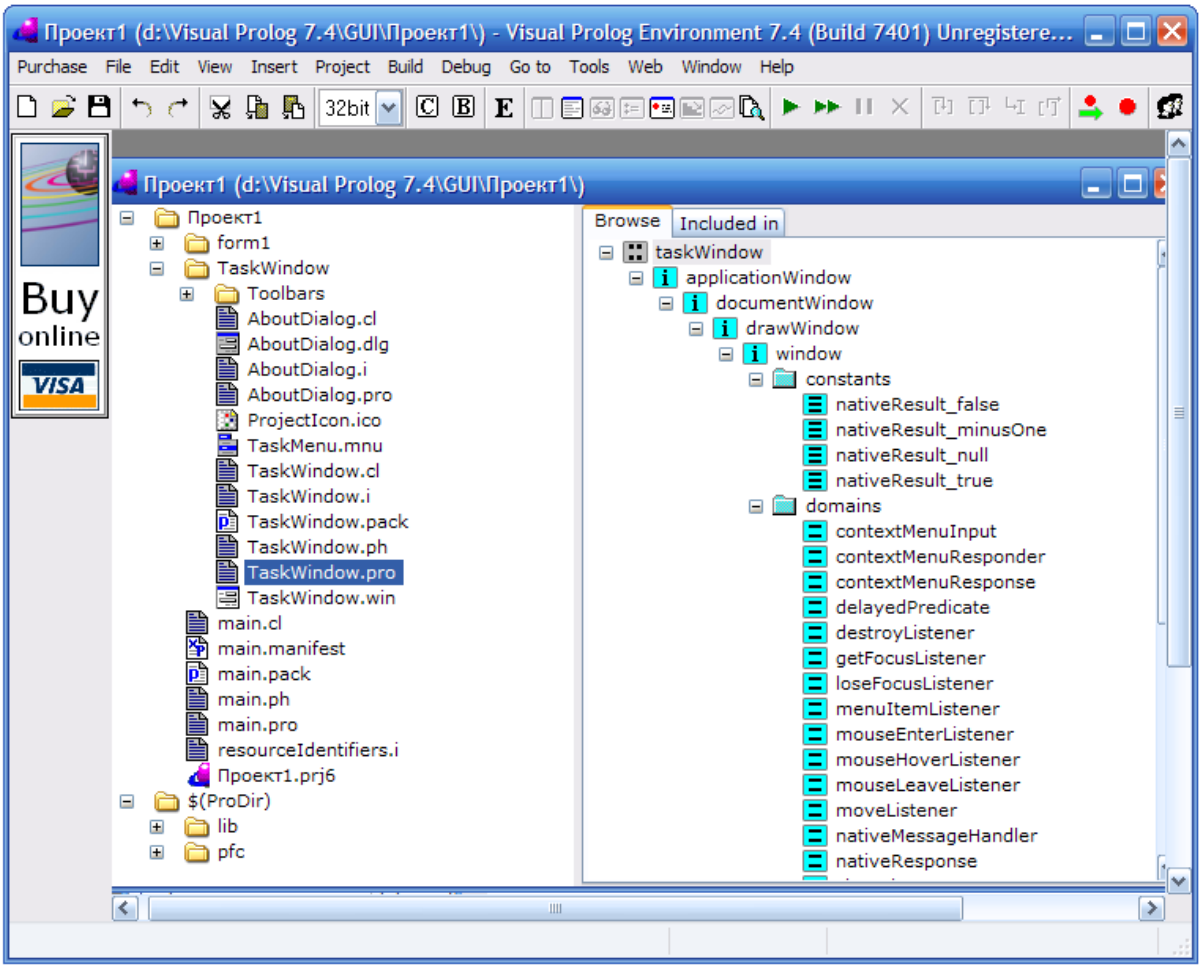


Рисунок 1.52. Завантаження TaskWindow.pro

Відкривається файл *TaskWindow*.pro.

Його вміст створюється автоматично під час компіляції, але за замовчуванням не прописано правило реагування на відкриття нового файлу (як показано на рис. 1.53):

```
clauses
onFileNew(_Source,MenuTag).
```

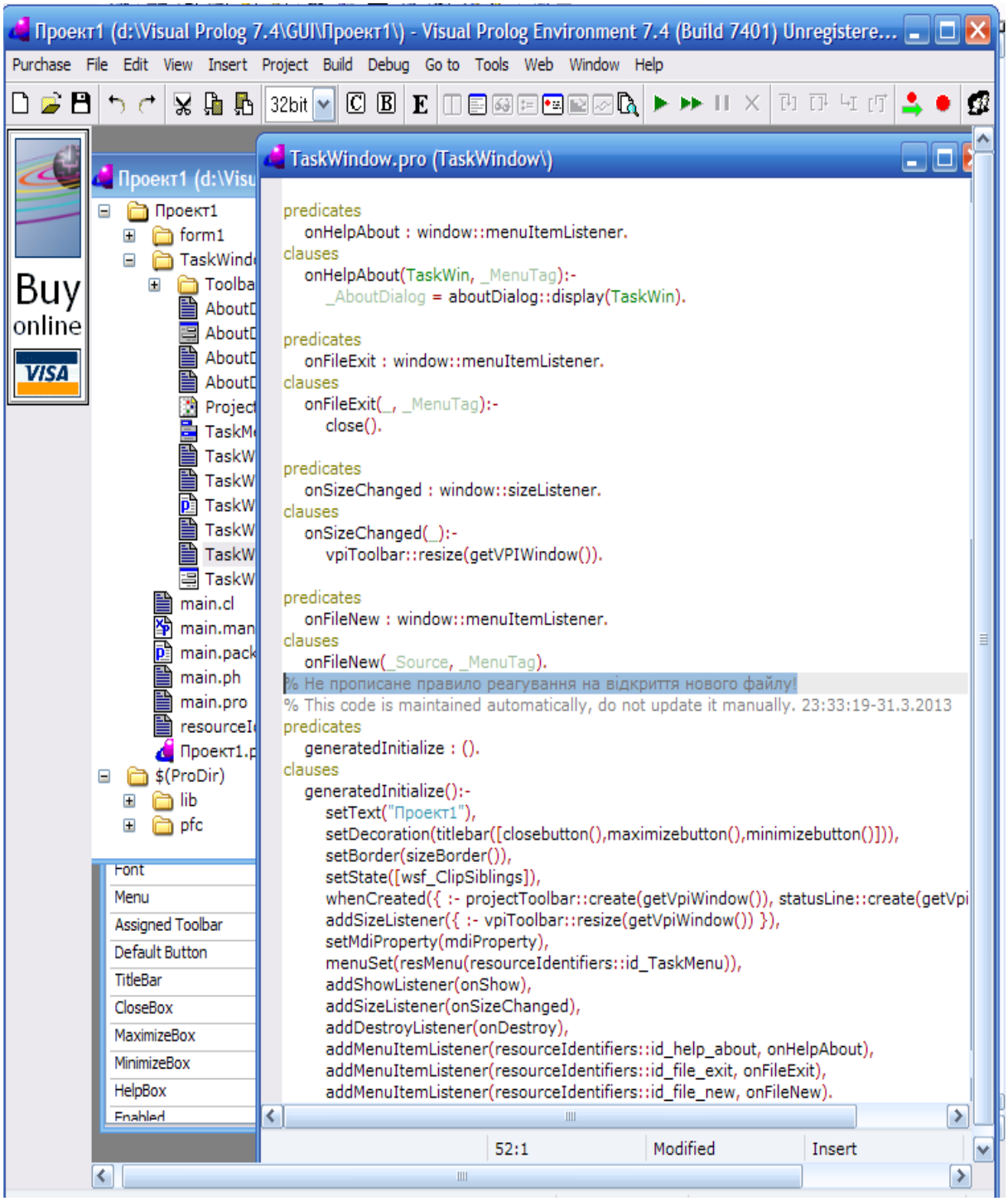


Рисунок 1.53. Код TaskWindow.pro

Слід дописати у відповідне місце (рис. 1.54) правило відображення форми *form1* за допомогою конструктора форми:

```
clauses
onFileNew(S,MenuTag):-
X= form1::new(S),X:show().
```

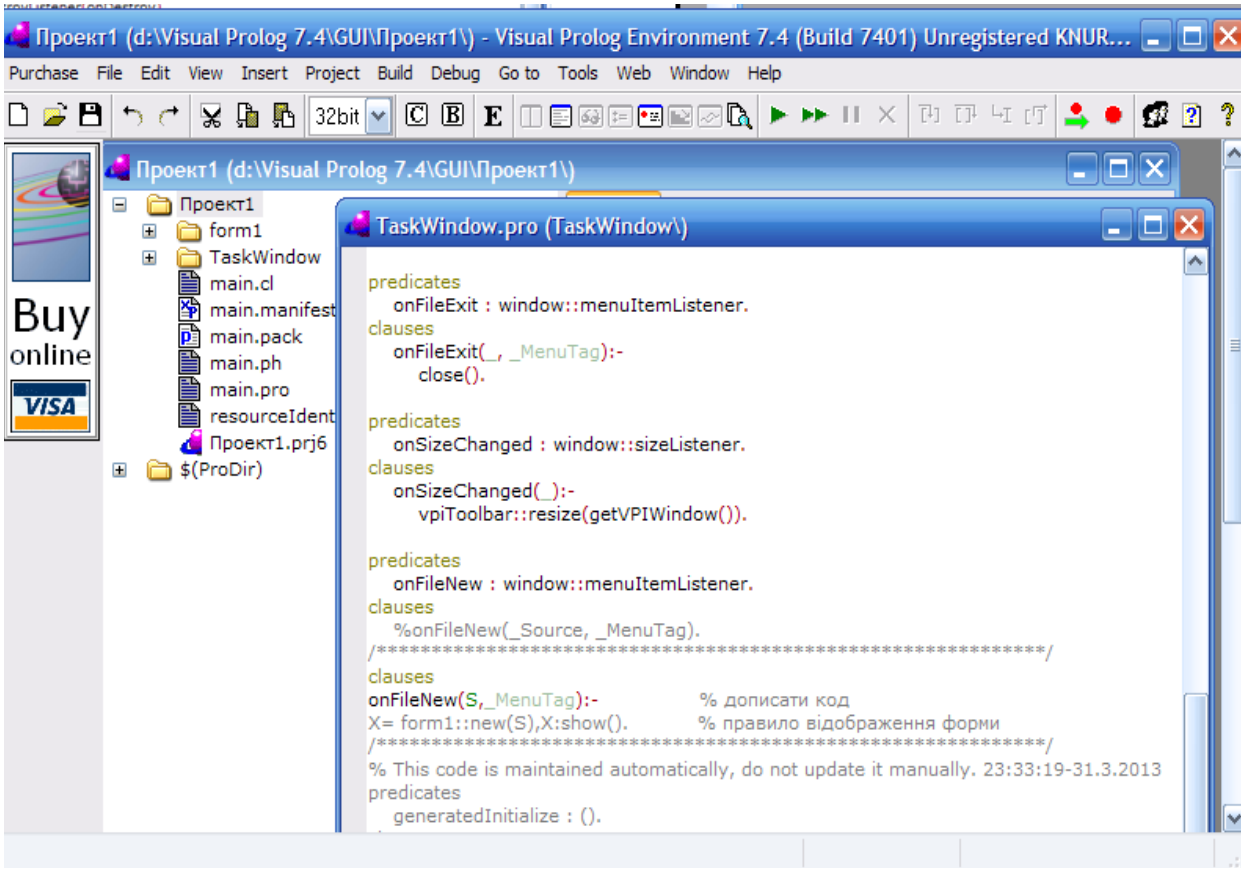


Рисунок 1.54. Додавання правила відображення форми form1

Далі слід виконати повторну компіляцію та виконання командою **Build | Execute**. В результаті буде відображатися вікно без форми. Команда **File | New** у вікні проекту приведе до відображення форми у вікні проекту (рис. 1.55).

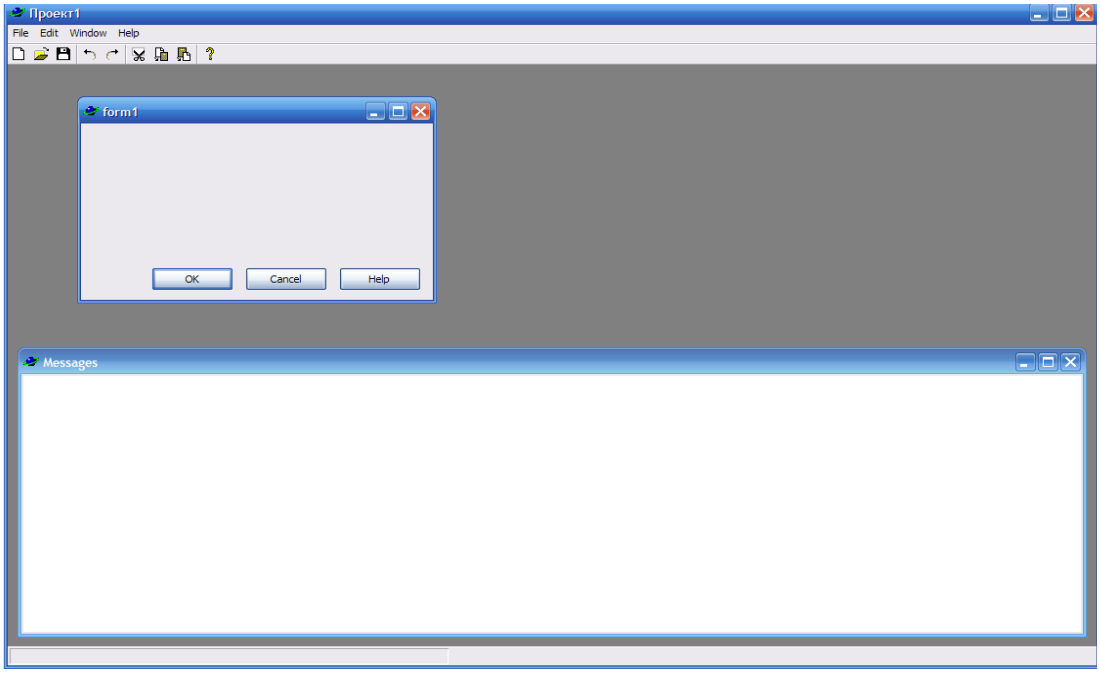


Рисунок 1.55. Відображення форми

Команда **File | New | OK** відобразить вікно форми (рис. 1.56).

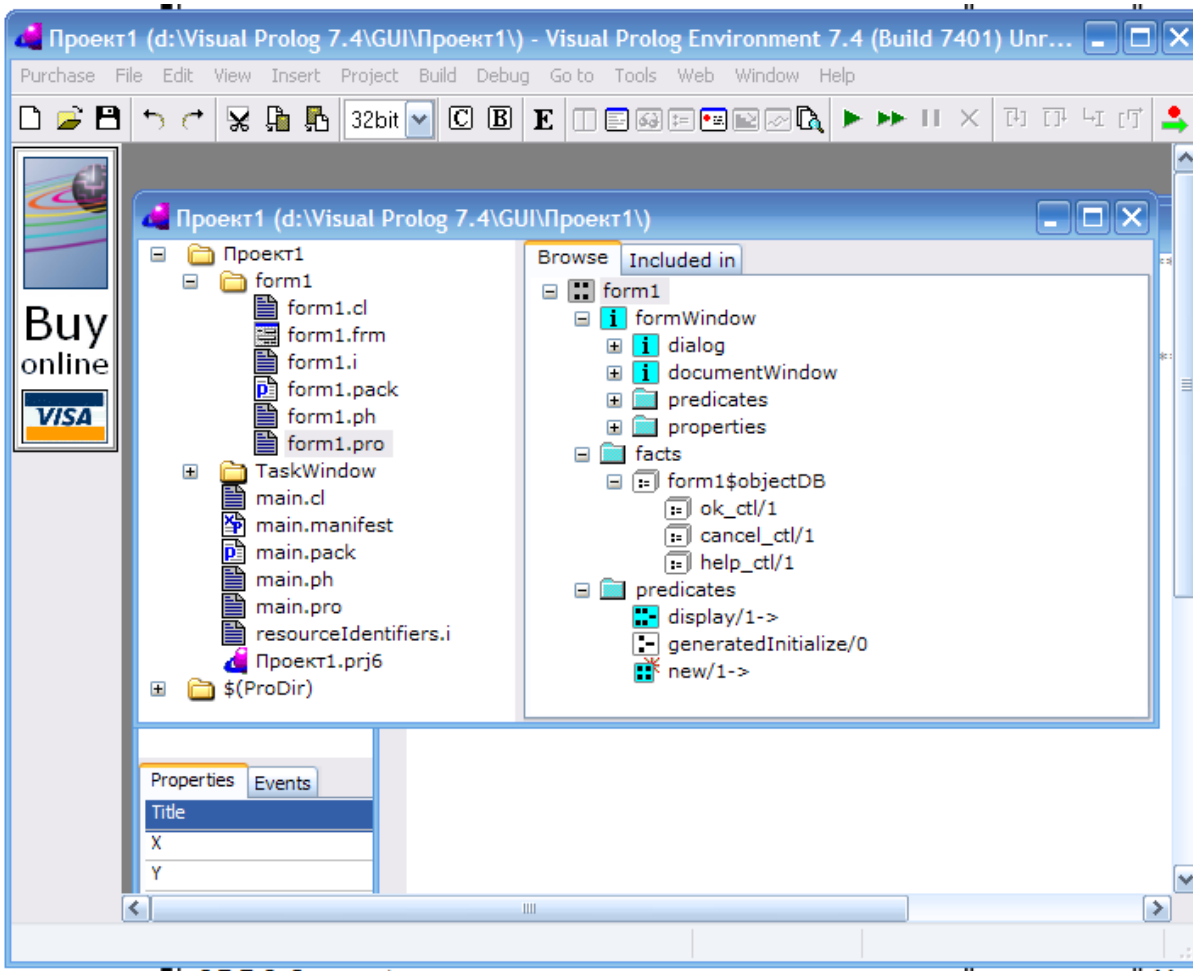


Рисунок 1.56. Відображення форми у вікні проекту

1.3. Можливості візуального середовища розробки Visual Prolog

Візуальне середовище розробки (VDE) дозволяє зручно створювати додатки, засновані на багаторівневій абстракції стандартних інтерфейсів користувача, забезпечених операційною системою, що їх підтримує.

1.3.1. Компілятор

Компілятор генерує код і виконує деякі дослідження загального аналізу потоків та перевірки детермінізму. Створює об’єктні файли для створення автономних виконуючих файлів або бібліотек. Виконує функцію розподілу регістрів та оптимізації хвостової рекурсії, додаткові перевірки для визначення проблем під час компіляції. Перевіряє факти ініціалізації у конструкторах. Забезпечує додатковий рівень захисту при програмуванні, вказує на помилки

невідповідності типів на рівні з значимими помилками, що гарантує цілісність програми впродовж всього життєвого циклу продукту.

Опції компілятора використовуються для встановлення параметрів компіляції стартового коду Прологу.

1.3.2. Експерт додатків (Application Expert)

Експерт додатків генерує та конфігурує нові проекти, враховуючи комбінації операційних систем; стратегію інтерфейсу користувача; властивості компонувальника зв'язків та інструментальних засобів, що використовуються; пакетів, що застосовуються тощо.

Завантажується командою меню **Project | New**.

Експерт додатків здатен підтримувати GUI application і Console application та підтримує декілька компіляторів C.

Вкладки вікна Project Settings дозволяють визначити властивості інтерфейсу візуального програмування для проекту. Всі властивості можуть бути змінені пізніше.

Панелі інструментів проекту, визначені за замовчуванням, створюються після створення проекту. Після визначення властивостей VPI генерують всі необхідні файли і створюють за замовчуванням додаток.

1.3.3. Експерт коду

Експерт коду – це інструмент, що сполучає код Прологу з розташуванням робочих вікон і вікон діалогів після їх проектування. Він управляє вікнами створення діалогу і обробки подій, генерує та підтримує Пролог-код, який керує використанням ресурсів.

Після розміщення компонентів ресурсів для генерації початкового коду використовують наступні експерти коду:

- експерт вікон і діалогових вікон;
- експерт пакету діалогових вікон;
- експерт панелей інструментів.

За їх допомогою виконується великий обсяг роботи, що значно заощаджує час для набору коду; користувач отримує у додатку готовий до роботи

інтерфейс користувача; надається стандартизований спосіб обробки ресурсів. Нові додатки створюються просто і швидко, підтримують внесення доповнень та змін.

Експерт коду Діалогу і вікна завантажується з вікна Проекту командою **Code Expert** або з меню **View**.

1.3.4. Дерево проекту

Управлінням компіляцією, компонуванням, зв'язуванням ресурсів тощо управляє інтегрований конструктор програм (make facility), шляхом перевірки мітки дати для перекомпілювання необхідних файлів. Залежності, для візуалізації структури проекту, можуть бути відображені у вигляді дерева проекту. Подвійний клік миші по імені файлу всередині дерева проекту викличе для цього файлу текстовий редактор.

1.3.5. Відладчик Visual Prolog

Відладка – це процес пошуку помилок у програмі. Відладчиком Visual Prolog є окремий додаток, який може бути завантаженим як з операційної системи, так і з VDE, який здатний відлагоджувати додатки 32- та 64-розрядної Windows в залежності від налаштувань. Він дозволяє відстежити виконання програми, встановити мітки зупинки та виконати програму по кроках. Відладчик також дає можливість перегляду змінних, що використовуються у додатку, який відлагоджують, перегляду фактів, подій VPI, використаної пам'яті тощо.

Відладчик працює з відкомпільованим кодом. При цьому Пролог пропонує засоби для обробки ситуацій, пов'язаних з появою помилок чи переривань, створених користувачем

Для активізації процесу відладки з VDE слід виконати команду меню **Builde | Execute** (або відповідна кнопка панелі інструментів).

Спочатку буде побудовано проект, далі буде завантажений цільовий додаток та активований відладчик Visual Prolog на цей додаток. Відладчик може завантажуватися під 32- чи 64-розрядну Windows для відладки програм користувацького інтерфейсу та консольних додатків.

Для завантаження виконуючої програми у відладчик слід визначити ім’я EXE-файлу та виконати команду Debug.

Команда **Run** виконується тільки для консольних додатків. Після завантаження програми, вся інша інформація по завантаженим модулям представлена у вікні Messages.

З метою перевірки виконання фрагменту програми або ізоляції помилок застосовується маркер зупинки Break Points, установку якого можна виконати з контекстного меню у рядку коду (при виборі команди Run програма буде виконуватись тільки до маркера зупинки). Маркерами зупинки, які можуть складати цілі списки, можна управляти (команда меню Debug), змінюючи їхні властивості (включено/виключено).

1.3.6. Інтегровані редактори для підготовки ресурсів

Дозволяють візуально проектувати та змінювати інтерфейс користувача в інтерактивному режимі. Елементи управління розміщують у діалогових вікнах за допомогою миші. Налаштування параметрів елементу виконується з контекстного меню. Рядки, курсори, піктограми, діалогові вікна є ресурсом, який необхідний для будь-якого додатку, що використовує графічний інтерфейс користувача GUI (Graphical User Interface).

Імпорт ресурсів здійснюється з динамічно зв’язаних бібліотек DLL (Dynamic Link Libraries), додатків, файлів ресурсів, з інших проектів Visual Prolog.

1.3.7. Текстовий редактор

Підтримує виділення кольором змінних, ключових слів, розрізняє типи аргументів предикатів, коментарі та інші елементи мови Visual Prolog. Підтримує команди копіювання(вирізування)/вставки (та виконання перелічених дій за допомогою миші), відміни/повтору дій, пошуку/заміни даних, підтримує гіперпосилання тощо.

Система підказок Visual Prolog основана на гіпертекстовій абстрактній машині, дозволяє вводити текст в інтерактивному режимі, створювати нові та здійснювати перехід по існуючим зв’язкам під час проектування (дозволяє

генерувати проміжні файли в стандартному для Windows-компіляторів форматі .RTF).

1.3.8. Браузер початкового коду

Компілятор Visual Prolog генерує інформацію для браузера вхідного коду, що дозволяє переглядати предикати модуля, глобальні предикати проекту, знаходить визначення та об'явлення предикатів і доменів, також класів і фактів.

Разом з Visual Prolog можна використовувати системи управління вхідним кодом, такі як Visual SourceSafe, PVCS, MKS. Visual Prolog дозволяє розділяти ресурси між різними проектами та одночасно виконувати їх різними програмістами.

1.3.9. Початковий код для інтерпретатора Пролог

У склад пакету Visual Prolog включено механізм логічного висновку (PIE – Prolog Inference Engine). Початковий код стандартного інтерпретатора Пролог написаний на Visual Prolog. Змінюючи його, можливо створювати власні спеціалізовані мови логічного програмування, механізми логічного висновку, оболонки експертних систем або програмні інтерфейси.

1.3.10. Інтерфейс візуального програмування(VPI)

Для графічних інтерфейсів користувача визначений переносимий інтерфейс прикладного програмування Application programming Interface Visual Prolog, що підтримується 32 та 64-розрядними ОС Windows. Visual Prolog працює під різними операційними системами та може генерувати програми для різних платформ.

Щоб забезпечити програмному застосуванню альтернативну роботу на різних платформах при застосуванні деяких непідтримуваних засобів і властивостей, виконують умовну компіляцію.

Компоненти (пакети) GUI передаються разом з початковим кодом та включають в себе таблицю, вікно дерева, вікно перегляду каталогів, панелі інструментів, діалогові вікна, засоби генерації звітів тощо.

1.3.11. Відкрита платформа

Мова Visual Prolog підтримує можливість обох парадигм – об'єктно-орієнтованого і декларативного програмування з підтримкою об'єктів і класів.

Visual Prolog здатен взаємодіяти з іншими програмними засобами. Може генерувати або самостійно визивати підпрограми, написані іншими мовами програмування (наприклад, Пролог використовує домени, що відповідають типам аргументів C. Якщо предикати Visual Prolog оголошуються в language C, то вони можуть визиватися безпосередньо з програм, написаних на мові C). Інтерфейс підтримує всі компілятори, що генерують стандартні модулі OBJ. Програми Visual Prolog можуть визивати бібліотеки DLL або самі можуть бути поміщеними в DLL.

1.3.12. Внутрішні бази даних

Visual Prolog має зручний засіб для розробки додатків баз даних. Система баз даних підтримує набір окремих упорядкованих послідовностей термів (від простих записів до дерев). Система управління базами даних може звертатися як до простих термів, так і виконувати пошук з поверненням по послідовностям термів для генерації чи підбору потрібних значень. Терми можуть бути збереженими у файлі або в пам'яті.

Підсистема баз даних підтримує B+ дерева, що забезпечують швидкий пошук даних та можливість ефективної зміни порядку термів.

При використанні підсистеми баз даних у мережевих додатках Visual Prolog дає можливість розділеного доступу до інформації у зовнішній базі даних, надаючи доступ до неї багатьом користувачам.

1.3.13. Зовнішні бази даних

Система зовнішніх баз даних являється швидким і найбільш гнучким способом зберігання великої кількості даних. У випадках використання даних, які вже існують в інших системах баз даних, або, якщо додатку треба розділяти дані з іншими додатками, що використовують якусь певну технологію баз даних, Visual Prolog надає можливість зв'язування з системами зовнішніх баз даних за допомогою переносимого SQL-інтерфейсу, заснованого на ODBC,

OCI-бібліотеках Oracle або на DB2. Visual Prolog також містить прямий інтерфейс з Microsoft ODBC API для платформи Windows.

1.3.14. Клієнт-серверна архітектура

Основним засобом для побудови додатків клієнт-серверного типу є зв'язування TCP/IP або NETDDE під Windows, що дає можливість передавати елементи Прологу будь-якої складності між різними процесами на одній машині або між програмами на різних машинах по мережі. Такі засоби дозволяють Прологу легко створювати логічні сервери або сервери баз даних.

Visual Prolog підтримує основні інтерфейси нижчого рівня та високорівневий інтерфейс, створюючи спрощене та безпечне використання сокетів (це API до протоколу TCP/IP, що використовується для встановлення зв'язків між програмами через Internet всередині мережі або між двома програмами на одному комп'ютері).

У Visual Prolog існує підтримка:

- FTP (використання передачі файлів Internet для відправлення/отримання їх з сервера Internet);
- HTTP (використання протоколу передачі гіпертекстової інформації у мережі Internet);
- CGI (створення програми, генеруючої динамічні Web-сторінки);
- ISAPI (сценарії на інформаційному сервері Microsoft або будь-якому іншому сервері HTTP, який підтримує інтерфейс ISAPI).

1.3.15. Інструментальні засоби обробки документів

Інструментальні засоби DOC_TOOL компанії PDC забезпечує обробку відформатованих документів, дозволяючи не залежати від формату представлення документу (.RTF, HTML чи IPF).

Існують конвертори з формату елементів Прологу у інші формати, а також синтаксичні аналізатори, що дозволяють конвертувати будь-який з цих форматів у формат елементів Прологу. Інструментарій відкриває такі можливості, як генерація документів Word, створення гіпертекстових документів тощо.

1.3.16. Особливості Visual Prolog Personal Edition

Комерційна версія Visual Prolog надає додаткові можливості у порівнянні з Personal Edition. Створення DLL у Visual Prolog можливе тільки для комерційної версії.

Visual Prolog Personal Edition надає наступні можливості:

- Development Environment (IDE) – середовище розробки (IDE);
- Text Editor – редактор тексту;
- Dialog Editor – редактор діалогу;
- Menu Editor – редактор меню;
- Toolbar Editor – панель інструментів редактора;
- Bitmap/Icon/Cursor Editor – редактор курсору;
- PFC GUI Support – підтримка PFC GUI;
- Standard GUI Controls, Windows, Dialog, Forms – стандартний інтерфейс управління вікон, діалогів, форм;
- Multithread Support – багато потокова підтримка;
- Basic VPI Library – основні VPI бібліотеки;
- Regular Expressions Support – регулярна підтримка виразів;
- Handle Templates Support – ручна підтримка шаблонів;
- Collections support – колекція підтримки;
- Mutable variables support – змінні підтримки;
- Migration Tool (for migrating Visual Prolog 5.2 project) – Limited support Download – для підтримки проектів Visual Prolog 5.2;
- Examples (Limited number of examples) – приклади (мінімальне число).

Запитання. Завдання

1. Вимоги для запуску VDE Visual Prolog.
2. З якого вікна Visual Prolog завантажуються пролог-програми на виконання чи компіляцію?
3. Характеристика пунктів меню Visual Prolog.
4. Яка інформація реєструється у вікні проекту?

5. Призначення вікна діалогу?
6. Створення файлу ініціалізації для Visual Prolog VDE?
7. Як створити проект?
8. Як налаштувати компілятор для проекту?
9. Як виконати перевірку налаштування системи VDE?
10. Як у середовищі VDE виконується тестування автономних виконуючих програм, які не підлягають тестуванню утилітою Test Goal?
11. Як створити виконуючу програму у середовищі VDE?
12. Які основні функції компілятора?
13. Призначення експерту додатків?
14. За допомогою чого виконується процес пошуку помилок у пролог-програмі?
15. У якому вікні відображається структура файлів завантаженої програми?
16. Призначення браузера початкового коду?
17. У якому форматі підтримує Пролог обробку відформатованих документів?
18. Де зберігаються терми (послідовності термів) після генерації та підбору потрібних значень?
19. Роль B+ дерев у підсистемі внутрішніх баз даних?
20. Взаємодія Visual Prolog з іншими програмними засобами?

РОЗДІЛ 2. ОСНОВИ МОВИ ПРОГРАМУВАННЯ VISUAL PROLOG

У розділі викладено теоретичні основи мови Пролог, що ґрунтуються на поняттях логіки предикатів першого порядку – такій системі у логіці, яка дає можливість виразити все, що відноситься до математичних логічних конструкцій і звичайної розмовної мови.

2.1. Логічні основи Прологу

Про мову програмування Пролог говорять як про формальну систему, тому що її фрази побудовані на основі чітких правил з умовою, щоб для будь-якої послідовності символів можливим було твердження: чи є вона правильно збудованою конструкцією цієї мови, чи ні. Логіка і теорія такої системи являється формалізацією мислення людини та представлення її знань із застосуванням аксіоматичного методу побудови наукової теорії, в якій за основу береться ряд основоположних (таких, що не потребують доказів), таких, що не потребують доказів, положень, – аксіом чи постулатів.

Формальна система будується як чітко визначений клас фраз та формул. У класі виділяється підклас теорем даної формальної системи. При цьому формули формальної системи не несуть у собі ніякого змістовного значення, вони створюються з довільних знаків або символів, виходячи тільки з міркувань зручності з вимогою найбільш адекватно та у повному обсязі відображати завдання і дедуктивну структуру математичної чи змістовної теорії.

Теоретичні основи мови Пролог ґрунтуються на поняттях логіки предикатів першого порядку (математична логіка). В математичній логіці, диз'юнкт Хорна (названі на ім'я логіка Альфреда Хорна) – це диз'юнкція літералів з не більше ніж одним позитивним літералом.

В логіці висловів літералом називають змінну або її логічне заперечення. Відповідно, позитивним літералом називають безпосередньо змінну, а негативним літералом – логічне заперечення зміни.

Диз'юнкт – це диз'юнкція скінченного числа літералів. Якщо диз'юнкт не містить літералів, його називають порожніми диз'юнктом. Диз'юнкт Хорна з

рівно одним позитивним літералом є визначеним диз'юнктом. Диз'юнкт Хорна без позитивних літералів іноді називається ціллю або запитом, зокрема в логічному програмуванні.

Алфавіт мови обчислення виразів складається з пропозиціональних змінних, логічних зв'язків і дужок. Правильно побудовані вирази мови обчислень фраз називаються формулами. Формула Хорна є кон'юнкція диз'юнктів Хорна, тобто формула в кон'юнктивній нормальній формі, всі диз'юнкти якої є хорновськими. Подвійними диз'юнктом Хорна називають диз'юнкцію з не більше ніж одним негативним літералом. Диз'юнкти Хорна грають основоположну роль в логічному програмуванні і мають зважений додаток в конструктивній логіці.

Основні логічні операції з логічними змінними (булевими змінними, які набувають одного з двох можливих значень істина або хибність):

- *заперечення* (протилежність);
- *кон'юнкція* (функція, яка приймає значення 1, якщо всі змінні дорівнюють 1, і рівна 0, якщо хоча б одна з цих змінних дорівнює 0. По застосуванню – логічне «*і*»);
- *диз'юнкція* (називається така функція, яка дорівнює 0, всі змінні дорівнюють 0, і дорівнює 1, коли хоча б одна змінна дорівнює 1. По застосуванню – логічне «*або*»);
- *імплікація* (імплікація як булева функція помилкова лише тоді, коли посилання істинне, а слідство помилкове. По своєму застосуванню наближена до виразу «*якщо ... то*»).

Знаки логічних операцій називають логічними зв'язками, а вирази, які отримують при правильному використанні логічних змінних та логічних зв'язків, – логічними або *пропозиціональними* (proposition) *формулами*. Логічні операції називають також логічними булевими функціями.

Наділення значення істинності пропозиціональним змінним називається інтерпретацією цих змінних. Під інтерпретацією формули розуміють наділення значення істинності змінним, що входять до цієї формули. Виявлення того

факту, що з множини висловлювань (формул обчислення) логічно слідує деяке інше висловлювання (формула) – є однією із задач обчислення.

Диз'юнкти Хорна пов'язані з доказом теорем через резолюції першого порядку, оскільки резолюція двох диз'юнктів Хорна є диз'юнктом Хорна. В математичній логіці і автоматичному доказі теорем, правило резолюцій – це правило виведення, що звертається до методу доказу теорем через пошук суперечностей, та використовується в логіці висловів і логіці предикатів першого порядку.

Правило резолюцій, застосоване послідовно для списку виразів, дозволяє відповісти на питання, чи існує в початковій множині логічних виразів суперечність. Резолюція цілі і визначеного диз'юнкта також є диз'юнктом Хорна. В автоматичному доказі теорем, це може давати велику ефективність в доведенні теореми, представленої у вигляді цілі.

Резолюція цілі з визначеним диз'юнктом для отримання нової цілі є основною для правила виведення, що використовується для реалізації логічного програмування і мови програмування Пролог. В логічному програмуванні визначений диз'юнкт використовується як процедура редукції цілі.

Наприклад, диз'юнкт Хорна працює як процедура: щоб показати *f*, слід показати *a, b, . . ., n*.

Щоб підкреслити це зворотнє застосування диз'юнкта, в математичній логіці часто використовується оператор (приналежність) ⟵:

f ⟵ *(a^b^…^n)*

На Пролозі це записується у вигляді правила, де символ «:-» означає «якщо…то»

f :- a, b, ..., n.

Пропозиціональні диз'юнкти Хорна також представляють інтерес для теорії складності обчислень, де задача пошуку множини істинних значень, виконуючих кон'юнкцію диз'юнктів Хорна, є повною.

Вважається, що задана якась формальна система F, якщо є визначений:

- *абетка системи* – множина символів;

- *формули системи* – деяка підмножина всіх слів, які можна утворити з символів, що входять в абетку (звичайно задається процедура, яка дозволяє складати формули з символів алфавіту системи);
- *аксіоми системи* – виділена множина формул системи;
- *правила виведення системи* – скінченна множина відносин між формулами системи.

Одним з правил виводу є правило підстановки, яке в неявному вигляді може бути присутнім у визначенні того, що являється правилом побудови виразів (пропозиціональної форми). Всі входження будь-якої пропозиціональної змінної можуть бути замінені будь-якою пропозиціональною формулою. У такому випадку аксіоми називають схемами аксіом, тому що вони визначають нескінченну множину формул.

Друге правило виводу – правило висновку, яке полягає у наступному міркуванні: якщо F та F → Q – це формули (теореми), що виводяться, то й Q є формулою, що виводиться (теорема обчислення виразів формальної теорії).

Мова логіки предикатів – одна з формальних мов, найбільш наближених до природної мови. Мова обчислення виразів є окремим випадком мови обчислення предикатів, тому вирази (пропозиціональні формули) являються формулами мови обчислення предикатів. Для того, щоб формально описати мову логіки як мову формальної теорії, необхідно задати множину його символів і правила побудови виразів – синтаксис мови, конструкціями якого є: логічні зв’язки; квантори; предметні змінні і константи; функціональні і предикатні символи; терми; вирази; атомарні формули; формули.

Абетку логіки першого порядку (логіки предикатів) складають наступні символи:

- змінні;
- константи;
- функціональні символи;
- предикативні символи;

- пропозиціональні константи (істина і хибність);
- логічні зв’язки (заперечення, кон'юнкція, диз'юнкція, імплікація);
- квантори (загальна назва для логічних операцій, обмежуючих область істинності якого-небудь предиката);
- допоміжні символи (,), (;), (:-).

Всякий предикативний і функціональний символ має певне число аргументів. Якщо предикативний (функціональний) символ має *n* аргументів, він називається *n-вмістним* предикативним (функціональним) символом.

Термом слід називати вираз, створений із змінних і констант, можливо, з використанням функцій, а точніше:

- всяка змінна або константа є терм;
- якщо *t1,...,tn* – терми, а *f* – *n*-вмістний функціональний символ, то *f(t1,...,tn)* – також є терм;
- інших термів немає.

По суті справи, всі об’єкти в програмі на Пролозі представляються саме у вигляді термів. Якщо терм не містить змінних, то він називається *основним* або *константним* термом. Атомна або елементарна формула створюється шляхом застосування предиката до термів, точніше, це вираз

p(t1,...,tn),

де p – n-вмістний предикативний символ, а t1,...,tn – терми.

Формули логіки першого порядку утворюються таким чином:

- всяка атомна формула є формула;
- якщо *A* і *B* – формули, а *X* – змінна, то вирази:

A, B (читається «A і B»);

A ; B (читається «A або B»);

-A (читається «*не A*» або «*заперечення*»);

- інших формул немає.

Літералом називають атомну формулу або заперечення атомної формули. Атом називається позитивним літералом, а його заперечення – негативним літералом.

Список аксіом і правил виведення логіки першого порядку буде приведений у відповідних розділах.

2.1.1. Теорія обчислення предикатів

Іноді, крім висловлювань стосовно властивостей об'єктів або відношень між об'єктами, треба мати ствердження, що будь-які об'єкти або деякі об'єкти володіють визначеними властивостями або знаходяться у деяких відношеннях. Різниця між поняттями «*відношення*» та «*предикат*» полягає у тому, що предикат являється «*індикатором*» відношення і може набувати значення «*істина*» або «*хибність*», в залежності від того, чи має місце це відношення. Двомісні відношення називають бінарними, одномісні властивості – унарними.

Квантор є загальною назвою для логічних операцій, що обмежують область істинності будь-якого предиката. Розрізняють *квантор загальності* (читається як «для всіх …», «для кожного …» або «кожен …», «будь-хто …», «для кожного …») та *квантор існування* (читається як «існує …» або «знайдеться …»).

Змінна, на яку накладається квантор, називається *зв'язаною*, при цьому число вільних змінних зменшується на одиницю. Вираз, на який накладається квантор, називається областю дії квантора. Вираз, який не містить вільних змінних, являє собою вираз, значення істинності якого не залежить від значень предметних змінних, що входять до нього. Перейменування зв'язаної змінної в області дії квантора не змінює значення істинності усього виразу, тому що значення істинності виразу залежить від значень вільних змінних, а не зв'язаних.

Формула обчислення предикатів, яка містить вільні змінні, називається виконуючою у даній інтерпретації, якщо при деякій підстановці предметних констант вона перетворюється в істинний вираз, інакше вона буде хибною в даній інтерпретації.

Формули обчислення предикатів діляться на три класи:

- *формули загального значення* – істинні в будь-яких інтерпретаціях;
- *протиріччя* – хибні в будь-яких інтерпретаціях – заперечення;

- *нейтральні формули* – істинні в одних і хибні в інших інтерпретаціях.

Обчислення предикатів, в яких є визначені функціональні, предикатні символи та предметні константи, називається прикладним обчисленням предикатів або теорією першого порядку. В якості предикатних символів можуть, наприклад, використовуватися предикати «=», «>», «<»; функціональних літер – знаки арифметичних операцій; предметних констант – натуральні числа.

2.1.2. Логічний висновок в обчисленні предикатів

Обчислення предикатів є невизначеним. Але, якщо формула обчислення предикатів є загального значення, то існує процедура визначення факту для деяких класів формул. Якщо ту ж процедуру примінити до формул, які не є загального значення, це може привести до зациклювання, тобто до виконання незкінченної послідовності операцій.

Для того щоб доказати, що з множини формул логічно виводиться дана формула, беруть заперечення цієї формули та додають її до вхідної множини формул. Після чого доказують протиріччя формули, що являється кон'юнкцією формул вхідної множини і даної – метод доведення від зворотного.

Метод резолюцій (в обчисленні предикатів) представляє собою процедуру спростування (доказ протиріччя) множини диз'юнктів, яка включає в себе уніфікацію. *Уніфікацією* називається процес підстановок з ціллю отримання відповідних резолюцій диз'юнктів, тобто пошук необхідної підстановки і доказ протиріччя даної множини диз'юнктів при даній підстановці. На кожному кроці алгоритму уніфікації для двох літералів приміняється підстановка до кожного з літералів. Якщо кількість літералів більше двох, то послідовно уніфікується перший з другим літералом, потім результат уніфікується з третім і т. д., поки не буде уніфікована вся множина літералів або отримано відповідь про те, що множина не може бути уніфікована. Уніфікація є дуже важливою складовою частиною метода резолюцій, а алгоритм уніфікації – одним з важливих механізмів мови логічного програмування.

Метод резолюцій розглядають як правило виводу в обчисленні, що реалізується програмою на мові логічного програмування Пролог. Правильно побудованими виразами (формулами) такого обчислення являються фрази (речення – диз'юнкти) та множини фраз. Резолютивним висновком є послідовність фраз, кожна з яких є фразою з множини фраз, або отримана з попередніх фраз за допомогою правила резолюції.

У Пролозі використовують метод резолюцій для хорновських диз'юнктів, які містять не більше одного позитивного літерала, що в Visual Prolog записується як:

F:– F1, F2, … , Fn.

Фрази такого вигляду називають у Пролозі *правилами*. У разі відсутності негативних літералів, фрази називають *фактами*. Множина правил і фактів складає програму – опис предметної області у вигляді множини аксіом прикладного обчислення предикатів. Фраза, що не містить позитивних літералів називається *ціллю* або *запитом* до програми та представляє собою заперечення теореми, що приміняється до множини фактів і правил. Виконання програми полягає у резолютивному висновку цієї теореми. До того ж вважається, що всі змінні у правилах замкнуті загальним квантором, а у запиті – квантором існування. При виконанні програми виконується уніфікація (пошук по зразку), властива результативному виведенню, у процесі чого виконується пошук значень змінних, що задовольняють запиту.

2.2. Основи мови логічного програмування

Ідея логічного програмування заключається у використанні комп'ютера для отримання висновків з декларативного опису предметної області, особливістю опису якої є достатньо спрощена можливість переходу від виразів природною мовою до фраз на обмеженій мові логіки предикатів першого порядку.

За основу декларативного опису предметної області прийнята спеціальна форма представлення у вигляді фактів і правил.

Диз'юнкція записана у вигляді імплікації (посилання імплікації записується справа, а висновок зліва від знаку операції), ігноруючи знаки логічних операцій, має вигляд:

A1, A2, …, An ← B1, B2, …, Bm (n≥0, m≥0)

і називається *клаузою*. З метою підвищення ефективності метода резолюцій, який використовується в якості методу автоматизованого доказу теорем, беруться лише формули (клаузи) при *n≤1*.

При *n=1* з попередньої формули буде отримана формула, яка називається клаузом Хорна (хорновською фразою, хорновським диз'юнктом):

A ← B1, B2, …, Bm

при *m>0* цю формулу називають правилом, при *m=0* – фактом (при цьому знак імплікації ігнорується).

При *n=0* та *m>0*, тобто коли ліва частина формули (висновок) пуста – формулу називають запитом ціллю (цільовим твердженням, запитом):

← B1, B2, …, Bm

Множина фактів і правил разом з запитом представляє собою логічну програму.

2.2.1. Предикати

У Пролозі програма є набором фактів з правилами, що забезпечують отримання висновків на основі цих фактів (декларативна мова). Мова базується на фразах Хорна, які є підмножиною формальної системи, так званої логіки предикатів.

У природній мові відношення встановлюється у фразі. Предикат є «*індикатором*» відношення і може приймати значення *Істина* або *Хибність* залежно від того, має місце чи ні дане відношення.

Предикат складається або тільки з імені, або з імені і наступної за ним послідовності аргументів, поміщеної в круглі дужки, синтаксично завершується символом крапки «.». Аргументом або параметром предиката може бути константа, змінна або складений об'єкт.

Відношення в Пролозі називається Предикатом.

Аргументи – це об’єкти, які зв’язуються цим відношенням.

Ім’я предиката розпочинатися рядковою літерою, після якої може розміщуватися будь-яка послідовність прописних і рядкових літер, цифр та символів підкреслення. В іменах не допускається наявність символів «пропуск», при необхідності вони заміняються символами підкреслення.

Таблиця 2.1. Символи, які дозволені для використання в іменах предикатів

Назва символів	Приклади символів
Прописні літери	A, B.,Z… А,Б,В,Г…
Рядкові літери	a, b .,z…. а,б,в,г….
Цифри	0, 1, …,9
Символ підкреслення	_

Версії Прологу, не дозволяють, щоб ім’я предиката починалося із прописної літери. Ім’я предиката може мати довжину до 250 символів. У іменах предикатів забороняється використовувати символ «пропуск», символ мінус «-», зірочка «*» та інші алфавітно-цифрові символи (такі як «%», «@» тощо). У таблиці 2.1 перераховані коректні імена Visual Prolog, що можуть включати символи.

Імена предикатів і аргументів можуть складатися з будь-яких комбінацій цих символів за умови, дотримання правил побудови відповідних імен. У табл. 2.2 приведені коректні і некоректні імена предикатів.

Таблиця 2.2. Імена предикатів в Visual Prolog

Коректні імена предикатів	Некоректні імена предикатів
Факт	[факт]
заробітна_плата	* заробітна плата *
людино_години	людино/години
періодЧас	період-час
Створити_Нове_Меню	Створити Нове Меню
список_з_5_по_10	5<список<10

Предикати можуть різнитися арністю.

Арність предиката – це кількість аргументів, які він містить.

Факти програми можуть містити два предикати з одним і тим же ім’ям, але відрізнятися арністю. У розділі *clauses* версії предикатів з одним ім’ям і різною

арністю повинні розміщуватися разом, за винятком наступного обмеження – різна арність завжди визначається як повна відмінність предикатів.

У факті

вивчати(ігор,мова).

відношення вивчати – це предикат, а об'єкти *ігор* та *мова* – це аргументи.

Аргументами предиката є константи:

відношення_що_об'єднує_два_аргумента(константа,константа).

В якості аргументу використовується *змінна* (змінні), наприклад, щоб вияснити ім'я студента Деркача, слід виконати запит до фактів (внутрішньої БД) з одною змінною «*Імя*» в якості аргументу:

```
goal
студент("Деркач",Імя,вік).
```

Або, щоб вияснити ім'я студента на прізвище Деркач і його вік, слід виконати запит з двома змінними «*Імя*» та «*Вік*»:

```
goal
студент("Деркач",Імя, Вік).
```

Приклад з використанням складеного об'єкту в якості аргументу предиката. Четвертим аргументом в предикаті використовується складений об'єкт «*дата*»

студент(прізвище,імя,стать,дата).

де «*дата*» – це *функтор*, який визначає вид складеного об'єкта (дата народження) та його аргументи.

Предикати можуть зовсім не мати аргументів, наприклад

виконати( ).

або

run.

Але використання таких предикатів обмежене. Пояснення таке: слід вияснити, наприклад, чи є в програмі фраза *run*, та якщо *run* – це заголовок правила, тоді можна обчислити дане правило.

Логіка предикатів була розроблена для найбільш простого перетворення принципів логічного мислення в форму запису. Пролог використовує переваги синтаксису логіки для розробки програмної мови.

У логіці предикатів у фразах, складених природною мовою, перш за все, вилучаються всі неістотні слова. Потім слід змінити (перетворити) ці фрази, ставлячи в них на перше місце відношення/властивість, а після нього – згруповані об'єкти. Надалі об'єкти стають аргументами, між якими встановлюється це відношення/властивість.

Visual Prolog містить деякі предикати, які обробляються компілятором спеціальним чином. Їх не можна перевизначити в Пролог-програмі, такі як:

assert	fail	save
asserta	findall	term_bin
assertz	format	term_repce
bound	free	term_str
chain_inserta	magrecv	trap
chain_insertafter	msgsend	write
chain_insertz	not	Writef
chain_terms	readterm	
consult	ref_term	
db_btrees	retract	
db_chains	retractall	

Приклади фраз, перетворених відповідно до синтаксису логіки предикатів наведені в табл. 2.3.

Таблиця 2.3. Складання фраз, відповідно логіки предикатів

Природною мовою	Мовою логіки предикатів
1	2
Мова логічного програмування.	*мова(логічне_програмування).*

Продовження таблиці 2.3

1	2
Декларативна мова.	*мова(декларативна).*
Ігору подобається мова логічного програмування.	*подобається(ігор, логічне_програмування).*

Назар студент 4-го курсу вивчає мову Visual Prolog.	*студент("Назар",4, "Visual Prolog").*
Згідно навчального плану за спеціальністю Логічне програмування викладають на 4-му курсі.	*навчальний_план("Логічне програмування",4).*
За навчальною програмою видом контролю з Visual Prolog є іспит.	*навч_програма("Visual Prolog",іспит).*
Студент 4-го курсу Назар вивчає Visual Prolog, результат контролю – іспит.	*студент("Назар",4, "Visual Prolog",іспит).*

2.2.2. Факти

У Пролозі факт представляє або *властивість* об'єкту, або *відношення* між об'єктами (у природній мові відношення/властивість встановлюється у фразі). Факт самодостатній і не вимагається додаткових відомостей для його підтвердження. Він може бути використаний як основа для логічного висновку.

Пролог включає механізм висновку, який заснований на порівнянні зразків. За допомогою підбору відповідей на запити він знаходить відому інформацію, що зберігається, і намагається перевірити істинність гіпотези, запрошуючи для цього інформацію, про яку вже відомо, що вона істинна.

У Пролозі описують *об'єкти* (objects) і *відносини* (relations), а потім описують *правила* (rules), при яких ці відносини є істинними.

Отже, у логіці предикатів, що використовується Прологом, відношення відповідає простій фразі – *факту*.

Природною мовою фраза:

Ігору подобається книга.

встановлює відношення між об'єктами *Ігор* та *книга*. Відношенням є предикат *подобається* і факт виглядає так:

подобається (ігор,книга).

Приклад фрази на природній мові з відношенням «*подобається*»:

Ігору подобається Ірина.
Ірині подобається Ігор.

Ігору подобається книга.

Використовуючи синтаксис Прологу маємо запис:

```
подобається (ігор, ірина).
подобається (ірина, ігор).
подобається (ігор, книга).
```

Факти, крім відносин, можуть виражати і *властивості*.

Приклад фраз «*Книга цікава*», «*Цікава гра*» або «*Гепард – кішка*», «*Кішка Соня*» записаних мовою Пролог з використанням властивості:

```
цікава (книга).
цікава (гра).
кішка (гепард).
кішка (соня).
```

2.2.3. Запити(цілі)

Задавши програмі Пролог одночасно декілька фактів, можна ставити питання, що стосуються відносин/властивостей між ними (Запити – query). Ґрунтуючись на заданих фактах

```
подобається (ігор, ірина).
подобається (ірина, ігор).
подобається (ігор, книга).
```

можна відповісти на питання про ці відносини на мові Пролог. Запит слід виконувати у розділі програми *goal* (програма, написана мовою Visual Prolog, може складатися всього з одного розділу *goal*):

1) Проста ціль «Чи Ігору подобається Ірина?», коли відомі аргументи предиката подобається:

```
goal
подобається (ігор,ірина).
```

Одержавши такий запит, Пролог відповість *Yes* (так), тому що Пролог має факт, підтверджуючий, що це дійсно так.

2) Ускладнимо питання «Хто подобається Ігору?»

```
goal
подобається (ігор,Хто).
```

Синтаксис запису не змінився, оскільки запит дуже схожий на факт. Але в якості другого аргументу використовується звичайна змінна *Хто* (ім’я змінної розпочинається з великої літери та виділяється кольором).

Пролог-програми – це обмежений набір заданих фактів і правил. Однією з найважливіших особливостей Прологу є те, що на додаток до логічного пошуку відповідей на поставлені питання, він може мати справу з альтернативами і знаходити всі можливі рішення. Замість звичайної роботи програми від початку до її кінця, Пролог може повертатися назад і переглядати більше одного шляху при рішенні всіх складових частин завдання.

Пролог завжди шукає відповідь на питання, починаючи з першого факту, і перебирає всі факти, поки вони не закінчаться. Одержавши запит про те, хто Ігору подобається, Пролог відповість:

```
Хто=ірина
Хто=книга
2 Solutions
```

оскільки йому відомо про такі факти.

3) Якщо поставити питання «Хто подобається Ірині?»

```
goal
подобається (ірина, Хто).
```

то відповідь буде такою:

```
Хто= ігор
1 Solutions
```

При виконанні наступного запиту «Ірині подобається все, що подобається Ігору»

```
goal
подобається (ірина, Що), подобається (ігор, Що).
```

відповідь буде такою:

```
Хто= ігор
Хто = ірина
Хто = книга
3 Solutions
```

оскільки відомо, що Ірині подобається Ігор, і що Ірині подобається те ж саме, що й Ігору, а Ігору подобається Ірина і книга.

4) Якщо поставити питання «Яка книга подобається Ігору?»,

```
goal
подобається (ігор, Що),книга(Що).
```

то Пролог не одержить рішення, оскільки в даному випадку не відомі факти про конкретну книгу, а він не може вивести висновок, заснований на невідомих

даних (в даних фактах не дано якого-небудь відношення або властивості, щоб визначити, чи є які-небудь об'єкти, які відносяться до книги).

Замість слова «*запит*» у пролог-програмах використовують більш загальне слово «*ціль*». Трактування запитів як цілей таке: коли Ви даєте Прологу запит, насправді Ви даєте йому ціль для виконання, тобто треба знайти відповідь на питання, якщо воно існує.

Цілі можуть бути *простими* і *складеними*.

Приклад простої цілі:

```
goal
подобається (ірина, Що).
```

Або, наприклад, ціль, що складається з двох частин, коли потрібно знайти загальне рішення двох цілей. Шукати рішення для такого запиту можна, задавши складену ціль:

```
goal
подобається (ірина, Що), подобається (ігор,Що).
```

Ціль, що складається з двох і більш частин, називається *складеною* ціллю, а кожна частина складеної цілі називається *підціллю*.

Для того, щоб розуміти текст пролог-програм, слід зазначити що у Пролозі зв'язування змінних із значеннями проводиться двома способами: на вході і виході (розділ *predicates*). Напрям, в якому передаються значення, вказується в *шаблоні потоку параметрів* (*flow pattern*) – надалі скорочено «*потік параметрів*».

Коли змінна передається у фразу, вона вважається вхідним аргументом і позначається символом (*i*). Коли ж змінна повертається з фрази, вона є вихідним аргументом і позначається символом (*o*).

Приклад 1

```
% Запит, що не відрізняється від факту
predicates
  подобається(symbol,symbol)- nondeterm (i,i)
clauses
  подобається (ігор, ірина).
  подобається (ірина, ігор).
  подобається (ігор, книга).

goal
  подобається (ігор, ірина).
% відповідь: Так  (підтвердження факту)
```

Приклад 2

```
% Використання у запиті  другим аргументом звичайної змінної
predicates
  подобається(symbol,symbol)- nondeterm (i,o)

clauses
  подобається (ігор, ірина).
  подобається (ірина, ігор).
  подобається (ігор, книга).

goal
  подобається (ігор, Хто).
% відповідь: Хто=ірина і Хто=книга
```

Приклад 3

```
% Використання у запиті першим аргументом звичайної змінної
predicates
  подобається(symbol,symbol)- nondeterm (o,i)
clauses
  подобається (ігор, ірина).
  подобається (ірина, ігор).
  подобається (ігор, книга).
goal
  подобається (Кому, книга).
% відповідь: Кому=ігор
```

Приклад 4

```
% Використання у запиті в якості аргументів змінних
%Кому хто подобається%
predicates
  подобається(symbol,symbol)- nondeterm (o,o)

clauses
  подобається (ігор, ірина).
  подобається (ірина, ігор).
  подобається (ігор, книга).

goal
  подобається (Кому, Хто).
/* відповідь: Кому= ігор  Хто= ірина
Кому= ірина  Хто= ігор
Кому= ігор  Хто= книга*/
```

Складені цілі можна використовувати для пошуку рішення, в якому обидві підцілі *A* і *B* істинні (*кон'юнкція* – логічне «*і*»), розділяючи підцілі комою. Також можливе рішення у тому випадку, якщо істинна або підціль *A*, або підціль *B* (*диз'юнкція* – логічне «*або*»), підцілі розділяють символом крапки з комою «;».

Для виведення додаткового тексту або результату пошуку у рішенні використовують стандартний предикат *write(аргументи)*. У наступному прикладі ціль можна вважати складеною.

Приклад 5.

```
% використання стандартного предикату write( )
predicates
  подобається(symbol,symbol) - nondeterm (o,o)

clauses
  подобається (ігор, ірина).
  подобається (ірина, ігор).
  подобається (ігор, книга).

goal
  write("Подобається: ","\t"),подобається (Кому, Хто).
/* відповідь:
Подобається:   Кому = ігор   Хто= ірина
               Кому= ірина  Хто= ігор
               Кому= ігор  Хто= книга*/
```

Для упорядкування виведення результатів пошуку використовується стандартний предикат *nl*, який виконує перехід на наступний рядок.

```
goal
  write("Подобається: ","\t"),подобається (Кому, Хто),nl.
  /* відповідь:
Подобається:
Кому = ігор   Хто = ірина
Кому = ірина   Хто= ігор
Кому = ігор   Хто = книга*/
```

Створення складеної цілі (кон’юнкція) «Ірині подобається все, що подобається Ігору».

```
% складена ціль (кон'юнкція)
predicates
подобається(symbol,symbol) - nondeterm (i,o), nondeterm (i,o)

clauses
  подобається (ігор, ірина).
  подобається (ірина, ігор).
  подобається (ігор, книга).

goal
  подобається (ірина, Хто), подобається (ігор, Що).
/* відповідь:
Хто = ігор   Що = ірина
Хто = ігор   Що = книга*/
```

Додамо в існуючу базу даних нових фактів. Складемо ціль «Ігору подобається щось, якщо воно гарне».

```
% складена ціль (кон'юнкція)
predicates
 подобається(symbol,symbol)- nondeterm (i,o)
гарна(symbol) - nondeterm (i)

clauses
 подобається (ігор, ірина).
 подобається (ірина, ігор).
 подобається (ігор, книга).
 гарна(книга).
 гарна(сумка).
 гарна(яхта).

goal
 подобається (ігор, Що), гарна(Що).
/* відповідь:
Що = книга*/
```

Тому що відомий тільки той факт, що Ігору подобається книга і що ця книга цікава. Але, якщо запитати Пролог «Що ж подобається Ірині, якщо це щось гарне»,

```
goal
 подобається (ірина, Що), гарна(Що).
/* відповідь:
No Solution */
```

то рішень не буде (Пролог виведе «*No Solution*») через відсутність фактів.

Замінимо розділовий знак між підцілями у попередньому прикладі на крапку з комою «;», тобто задамо диз'юнкцію

```
goal
 подобається (ігор, Що); гарна(Що).
/* відповідь:
Що = ірина
Що = книга     %або
Що = книга
Що = сумка
Що = яхта
5 Solution*/
```

Перевіримо ще раз, як працює складена ціль «Хто подобається Ігору або Хто подобається Ірині ?» на існуючих фактах

```
predicates
 подобається(symbol,symbol) - nondeterm (i,o), nondeterm (i,o)
clauses
```

```
  подобається (ігор, ірина).
  подобається (ірина, ігор).
  подобається (ігор, книга).

goal
  подобається (ігор, Хто); подобається (ірина, Хто).
/*відповідь:
Хто=ірина
Хто=книга          %або
Хто=ігор
3 Solution*/
```

Для більшої наочності результатів пошуку додамо у ціль предикат *write( )*. Перепишемо ціль.

```
goal
  write("Ігору подобається: "),nl, подобається (ігор, Хто);
  write("Ірині подобається: "),nl, подобається (ірина, Хто).
/*Результат:
Ігору подобається:
Хто = ірина
Хто = книга
Ірині подобається:
Хто = ігор
3 Solutions*/
```

Слід звернути увагу на розділові знаки: у підцілях використовується кон'юнкція, а між підцілями – диз'юнкція.

2.2.4. Правила

У Пролозі можна судити про достовірність чого-небудь, логічно вивівши факт з інших фактів. Реалізує таку можливість конструкція Прологу – *Правило*.

Правило – це властивість або відношення, яке достовірно, коли відомо, що ряд інших відносин достовірні, оскільки висновок заснований на фактах. Синтаксично ці відносини розділені комами.

У Пролозі всі правила мають дві частини: *заголовок* і *тіло*, розділені спеціальним знаком «*:-*», що трактується як «*якщо*» (прототип оператора if… then). Тільки висновок про істинність заголовка правила Прологу виконується, за умови

ЗАГОЛОВОК істинний, якщо ТІЛО — істинне (або: якщо ТІЛО може бути виконане)

Узагальнений синтаксис:

Заголовок: - <Підціль>, <Підціль>, ..., <Підціль>.

Заголовок – це факт, який був би істинним, якби були істинними декілька умов. Його називається висновком або залежним відношенням.

Тіло – це ряд умов, які повинні бути істинними, щоб Пролог міг довести, що заголовок правила істинний. Тіло правила складається з однієї або більш підцілей. Підцілі розділяються комами, визначаючи кон'юнкцію, а за останньою підціллю правила слідує крапка.

Факти і правила – практично одне і те ж, крім того, що факти не мають явного тіла. Факти поводяться так, якби вони мали тіло, яке завжди істинне.

Пролог завжди шукає рішення, починаючи з першого факту чи/або правила, і проглядає увесь список фактів чи/або правил до кінця.

Механізм логічного виведення Прологу бере умови з правила (тіла правила) і проглядає список відомих фактів і правил, намагаючись задовольнити умовам. Якщо всі умови істинні, то залежне відношення (заголовок правила) вважається істинним, тобто правило – це висновок. Якщо всі умови не можуть бути узгоджені з відомими фактами, то правило нічого не виводить.

Для успішного вирішення правила Пролог повинен вирішити всі його підцілі і створити послідовний список змінних, належним чином зв'язавши їх. Якщо ж одна з підцілей помилкова, Пролог повернеться назад для пошуку альтернативи попередньої підцілі, а потім знов рушить вперед, але вже з іншими значеннями змінних. Цей процес називається *пошук з поверненням.*

Правило, яке визначає стан, коли відношення є істинним, звичайною мовою виглядає так:

Ігору подобається книга, якщо книга цікава.

Мовою предикатів:

подобається(ігор, книга):- цікава(книга).

Для фактів

подобається (ігор, ірина).
подобається (ірина, ігор).
подобається (ігор, книга).

Задаємо ціль у розділі *goal* «Ірині подобається все, що подобається Ігору», що відповідає відношенню «*подобається*»:

```
goal
  подобається (ірина, Що), подобається (ігор,Що).
```

Пролог знаходить рішення:

Ірині подобається Ірина та *Ірині подобається книга.*

Для фактів:

```
цікава (книга).
цікава (гра).
гарна(книга).
```

Складаємо ціль, відповідну відношенню «*подобається*»:

«Ірині подобається все цікаве»

```
goal
  подобається (ірина, Що), цікава (Що).
```

Знаходимо, що Ірині подобається все цікаве (*книга*, *гра*).

Для виведення правил в Пролозі потрібно змінити синтаксис:

```
подобається (ірина,Що):- подобається (ігор,Що).
подобається (ірина,Що):- цікава(Що).
```

Правило можна розглядати як процедуру, тобто вищеописані правила означають:

1) «Щоб довести, що Ірині подобається щось, доведіть, що це подобається Ігору».

2) «Щоб довести, що Ірині подобається щось, доведіть що це щось цікаве».

З такої точки зору правила можуть вказати Прологу виконати інші дії, відмінні від доказів фактів, наприклад, надрукувати що-небудь:

```
подобається (ігор,Що):- гарна(Що),write("Ігору подобається").
```

Приклад 1.

```
predicates
  подобається(symbol,symbol)- nondeterm (i,o), nondeterm (i,i)
  гарна(symbol) - nondeterm (i)

clauses
  подобається (ігор, ірина).
  подобається (ірина, ігор).
  подобається (ігор, книга).
  подобається (ірина,X):- подобається (ігор,X).
  гарна(книга).
  гарна(сумка).
  гарна(яхта).
```

```
goal
  подобається (ірина,Що).
/* відповідь:
Що = ігор
Що = ірина
Що = книга
3 Solutions*/
```

Перевіримо інше правило «Ігору подобається щось, якщо це щось гарне». Для цілі

```
goal
результат(ігор,Що).
```

виконаємо правило

```
результат(ігор,X):- подобається(ігор,X), гарна (X).
```

Результати пошуку фактів Прологом будуть такими:

```
Що=книга
1 Solution
```

Тому що тільки для одного об’єкту, який подобається Ігору, існує факт про те, що цей об’єкт гарний.

Приклад 2.

```
% на побудову правил
predicates
  подобається(symbol,symbol) - nondeterm(o,o)
  гарна(symbol) - nondeterm (i)
  зелений(symbol) - nondeterm (i)
  синій(symbol) - nondeterm (i)
  результат(symbol,symbol) - nondeterm(o,o)

clauses
  подобається(ігор,ірина).
  подобається(ірина,ігор).
  подобається(ігор,машина).
  подобається(ірина,камінь).
  подобається(ірина,сумка).
  гарна(машина).
  гарна(сумка).
  гарна(яхта).
  зелений(камінь).
  зелений("м'яч").
  синій(камінь).
  % Кому подобається щось, якщо те, щось - гарне.
  результат(Y,X):-подобається(Y,X),гарна(X).

goal
  % простий запит
  результат(Хто,Що).
```

```
/* відповідь:
Хто=ігор, Що=машина
Хто=ірина, Що=сумка
2 Solutions */
```

Якщо для тих же фактів задати ціль

```
goal
результат(ірина,Що).
```

та побудувати правило «Ірині подобається щось, якщо те, щось – гарне»

```
результат(Хто,Що):-подобається(Хто,Що),гарна(Що).
```

результатом буде відповідь:

```
Що=сумка
1 Solution
```

Якщо у розділі *goal* записати предикат ***виконати***, який не має аргументів, то для виведення результатів пошуку у правилі слід використовувати предикат *write( )*:

```
predicates
  подобається(symbol,symbol)- nondeterm (i,o)
  гарна(symbol) - nondeterm (i)
  зелений(symbol) - nondeterm (i)
  синій(symbol) - nondeterm (i)
  виконати() - nondeterm ()

clauses
  подобається(ігор,ірина).
  подобається(ірина,ігор).
  подобається(ігор,машина).
  подобається(ірина,камінь).
  подобається(ірина,сумка).
  гарна(машина).
  гарна(сумка).
  гарна(яхта).
  зелений(камінь).
  зелений("м'яч").
  синій(камінь).
  %Ірині подобається щось, якщо те щось - гарне.
  % якщо не вивести write(X)- Пролог напише відповідь Yes
  виконати:-подобається(ігор,X),гарна(X),write(X),nl.

goal
  виконати.
/* відповідь:
машина
yes */
```

2.2.5. Фрази

Факти і правила, відносини, властивості, основні конструкції і запити – всі ці терміни є частиною логіки і природної мови. Програми на мові Пролог складаються з двох типів фраз: фактів і правил, які називають фразами.

Використовуючи терміни Прологу, розглянемо фрази, предикати, змінні і цілі. По суті, є тільки два типи фраз, що становлять мову Прологу: фраза може бути або фактом, або правилом. Ці фрази в Пролозі відомі під терміном *clause*.

Ядро програм на Пролозі складається з фраз, як наведено у наступному прикладі. До відомих фактів поставлена ціль результат (*Викладач*, *Предмет*), після пошуку буде виведений список викладачів та назва дисциплін, які вони викладають. Правило

```
результат (Викладач,Дисципліна):- викладає (Викладач,Дисципліна),
навч_план (Дисципліна,'2').
```

уточнює умови пошуку, що природною мовою означає «вивести список викладачів та назву дисциплін, які вони викладають, якщо за навчальним планом ці дисципліни викладають на 2-му курсі».

```
predicates
 студент(string,char)-nondeterm(o,o)
 викладає(string,symbol)-nondeterm(o,o)
 навч_план (symbol,char)-nondeterm(i,i)
 результат(string,symbol)-nondeterm(o,o)

clauses
 результат (Викладач,Дисципліна):-викладає(Викладач,Дисципліна),
 навч_план (Дисципліна,'2').
 студент("Орлов",'4').
 студент("Бабенко",'4').
 студент("Кривцун",'3').
 студент("Дяченко",'4').
 студент("Сай",'3').
 студент("Палій",'2').
 студент("Дяченко",'1').
 студент("Чак",'5').
 студент("Жушман",'4').
 студент("Старик",'5').
 студент("Палій",'1').
 студент("Кривцун",'2').
 студент("Кулик",'4').
 студент("Набока",'3').

 викладає("Девяткін",функціональне_програмування).
 викладає("Кнуренко",логічне_програмування).
 викладає("Кнуренко","комп'ютерна_інженерія").
```

```
викладає("Піддубний",вища математика).
викладає("Ульянова","об'єктно_орієнтоване_програмування").
викладає("Ульянова",штучний_інтелект).
викладає("Піддубний",теорія_ймовірності).
викладає("Рудяков",схемотехніка).
викладає("Рудяков",фізика).

навч_план(функціональне_програмування,'3').
навч_план (функціональне_програмування,'4').
навч_план (логічне_програмування,'4').
навч_план ("комп'ютерна_інженерія",'1').
навч_план (вища_математика,'1').
навч_план ("об'єктно_орієнтоване_програмування",'2').
навч_план ("об'єктно_орієнтоване_програмування",'3').
навч_план ("об'єктно_орієнтоване_програмування",'4').
навч_план (штучний_інтелект,'4').
навч_план (теорія_ймовірності,'2').
навч_план (схемотехніка,'3').
навч_план (фізика,'2').
навч_план (фізика,'1').

goal
 результат (Викладач,Предмет).
```

Результатом пошуку є:

```
Викладач=Ульянова, Дисципліна=об'єктно_орієнтоване_програмування
Викладач=Піддубний , Дисципліна=теорія_ймовірності
Викладач= Рудяков, Дисципліна=фізика
3 Solutions
```

2.2.6. Змінні

У Пролозі змінні дозволяють записувати загальні факти і правила та ставити загальні питання (*цілі*).

При використанні у Пролог-програмах змінних, перший символ імені повинен бути прописною літерою або символом підкреслення, після якого може стояти будь-яка кількість літер (прописних або рядкових), цифр або символів підкреслення, та не містити символу «пропуск».

Зручно використовувати в назві змінної літери різного регістра:

```
ПравильнеІм'яЗмінної
РахунокДоходу_Витрати
Розпродажі_25_II_12
```

Наступні три – неправильні:

```
1текст
Друга книга
"текст"
```

Для зручності рекомендовано змінним привласнювати змістовні імена (наприклад, *Особа*, *Ціна*, *Кількість_приходу*, *Група*, *Книга* та ін.). Осмислений вибір імен змінних робить програму зручнішою для читання.

Наприклад, *подобається(Людина,футбол)* *значно* краще, ніж *подобається(X,футбол)*, тому що змінна *Людина* несе більше інформації, ніж *X*.

Розглянемо використання змінних на прикладі.

```
predicates
  дисципліна(string,symbol,symbol) - nondeterm (o,i,i)
  корпус(string,string) - nondeterm (i,i)
  дата (string,string) - nondeterm (o,o)
  правило_1(string) - nondeterm (o)

clauses
  дисципліна ("Комп'ютерна_графіка",лекція,непарний).
  дисципліна ("Комп'ютерна_графіка",практика,парний).
  дисципліна ("Моделювання_систем",лекція,непарний).
  дисципліна ("Моделювання_систем",практика,парний).
  дисципліна ("Комп'ютерні_мережі",лекція,непарний).
  дисципліна ("Фінансовий_менеджмент",симінар,парний).
  дисципліна ("Логічне_програмування",лекція,непарний).
  дисципліна ("Логічне_рограмування",лабораторна_робота,непарний).
  дисципліна ("Комп'ютерні_мережі",практика,парний).

корпус ("Комп'ютерна_графіка","К1").
корпус ("Моделювання_систем","К2").
корпус ("Комп'ютерні_мережі","К1").
корпус ("Фінансовий_менеджмент","К2").
корпус ("Логічне_програмування","К1").

  дата("Теорія_автоматичного_управління","Вівторок_4_пара").
  data ("Комп'ютерні_мережі","Четверг_4_пара").
  data ("Логічне_програмування","Понеділок_4_пара").

%Вивести список дисциплін, якщо вид занять є лекція,
% яка викладається на непарному тижні в корпусі К1
  правило_1(X):-
  дисципліна (X,лекція,непарний),
  корпус (X,"К1").

goal
  правило_1(Дисципліна).
```

У наведеному прикладі для виведення списку дисциплін, що відповідає певним критеріям, побудоване правило, у якому *X* – це змінна. Змінна, яка розміщена у запиті, має ім’я *Дисципліна* (змістовна змінна, що полегшує читання програми).

У знайдених Прологом результатах, відображається змінна *Дисципліна*.

```
Дисципліна =Комп'ютерна_графіка
Дисципліна =Комп'ютерні_мережі
Дисципліна =Логічне_програмування
3 Solutions
```

2.2.7. Анонімні змінні

Якщо слід знайти тільки певну інформацію по запиту та ігнорувати всю іншу, то можна використовувати *анонімні* змінні. У Пролозі анонімні змінні позначаються символом підкреслення «_» та зіставляються з будь-якими даними.

Використання такої змінної демонструє наступний приклад. Якщо до фактів програми задати ціль, яка повинна виводити список усіх дисциплін, та кількість відведених для них годин за навчальним планом, без різниці це лекція, семінар, лабораторна робота чи практика

```
predicates
  видконтролю(string,string,word)-nondeterm(o,o,o)

clauses
  видконтролю("менеджмент","залік",90).
  видконтролю("моделювання систем","залік",108).
  видконтролю("теорія автоматичного управління","залік",90).
  видконтролю("технологія програмування","залік",90).
  видконтролю("фінансовий менеджмент","залік",108).
  видконтролю("комп'ютерна графіка","залік",90).
  видконтролю("логічне програмування","екзамен",90).
  видконтролю("основа охорони праці","екзамен",54).
  видконтролю("системний аналіз","екзамен",144).
  видконтролю("комп'ютерні мережі","екзамен",126).

goal
  видконтролю(Дисципліна,_,Всього_годин).
```

то результатаом пошуку буде список:

```
Дисципліна=менеджмент, Всього_годин=90
Дисципліна=моделювання систем, Всього_годин=108
Дисципліна=теорія автоматичного управління, Всього_годин=90
Дисципліна=технологія програмування, Всього_годин=90
Дисципліна=фінансовий менеджмент, Всього_годин=108
Дисципліна=комп'ютерна графіка, Всього_годин=90
Дисципліна=логічне програмування, Всього_годин=90
Дисципліна=основа охорони праці, Всього_годин=54
Дисципліна=системний аналіз, Всього_годин=144
Дисципліна=комп'ютерні мережі, Всього_годин=126
10 Solutions
```

Анонімна змінна може бути використана на місці будь-якої іншої змінної і їй ніколи не привласнюється значення.

Наприклад, в наступному запиті знадобиться дізнатися, які люди є батьками, але не цікаво, хто є їхні діти. Пролог знає, що кожного разу, коли використовується символ підкреслення в запиті, не потрібна інформація про значення, представлене на місці анонімною змінною.

```
predicates
  чоловік(symbol)
  жінка(symbol)
  батьки(symbol, symbol)

clauses
  чоловік(іван).
  чоловік (ігор) .
  жінка(ліна).
  жінка(юлія).
  батьки(іван,ігор).
  батьки(ліна,ігор).
  батьки(ігор, юлія).

goal
  батьки(Батьки,_).
```

Одержавши такий запит, Пролог (Test Goal) відповідає:

```
Батьки=іван
Батьки=ліна
Батьки=ігор
3 Solutions
```

У цьому випадку Пролог знаходить і видає троє батьків, але він не виводить значення, пов’язані з другим аргументом у фактах *батьки*.

Наступний приклад демонструє, як з існуючих фактів слід вивести прізвища студентів та ще деяку інформацію, ігноруючи деякі дані.

Результати вивести за прикладом:

Список 1-й: «Студент чоловічої статі, старшокурсник, місце проживання – не важливо». Наприклад

```
Іван Горкін - працевлаштування (поважна)
.....
```

Список 2-й: «Студент, стать жіноча, місцева, курс не має значення, неповажні причини». Наприклад

```
Інна Пронь - не цікаво
```

Список 3-й: «Студент, стать жіноча, місцевий, з будь-якого курсу, неповажна причина, кількість пропущених годин. Наприклад

Інна Пронь незацікавленість у предметі=25год.
……

```
/*ППРИКЛАД: Аналіз причин пропусків занять студентами */
predicates
список()- nondeterm ()
інфо(string,string,string,integer)- nondeterm (i,i,i,o)
причина(string, symbol) - nondeterm (i,i),(o,i)
поваж_працевл(string,integer) - nondeterm (o,i)
студент(string,string,symbol,symbol,integer)- nondeterm (o,o,i,o,i),(o,o,i,i,o)
положення1(string, symbol)- nondeterm (i,i)
положення2(string, symbol) - nondeterm (i,i)

clauses
студент("Іван","Горин",чол,ін,5).
студент("Ігор","Санін",чол,м,2).
студент("Андрій","Кокін",чол,ін,1).
студент("Євген","Кокін",чол,м,5).
студент("Ірина","Пронь",жін,ін,4).
студент("Інна","Пронь",жін,м,1).
студент("Світлана","Моря",жін,ін,1).
студент("Світлана","Моря",жін,ін,4).
студент("Євгенія","Меньша",жін,ін,5).
студент("Олена","Чайка",жін,м,3).
студент("Іван","Ванін",чол,м,4).
студент("Інна","Мирна",жін,ін,2).
студент("Ігор","Сірий",чол,ін,4).

причина("хвороба", пов).
причина("несприятливі природні умови", пов).
причина("олімпіада", пов).
причина("конференція", пов).
причина("стомленість", непов).
причина("погане самопочуття", непов).
причина("незацікавленість у предметі", непов).
причина("набридло", непов).
причина("недобрий ранок", непов).
причина("працевлаштування",пов).

поваж_працевл("працевлаштування",4).
поваж_працевл("працевлаштування",5).

положення1("по догляду за дитиною", жін).
положення2("несприятливі природні умови", ін).

інфо("Іван","Горін","недобрий ранок",2).
інфо("Іван","Горін","працевлаштування",81).
інфо("Іван","Горін","конференція",6).
```

```
інфо("Іван","Горін","хвороба",36).
інфо("Ігор","Санін","набридло",60).
інфо("Андрій","Кокін","хвороба",18).
інфо("Євген","Кокін","конференція",6).
інфо("Ірина","Пронь","стомленість",18).
інфо("Інна","Пронь","незацікавленість у предметі",25).
інфо("Світлана","Моря","олімпіада",48).
інфо("Світлана","Моря","працевлаштування",90).
інфо("Євгенія","Меньша","хвороба",48).
інфо("Олена","Чайка","набридло",24).    %
інфо("Іван","Ванін","працевлаштування",90).
інфо("Інна","Мирна","по догляду за дитиною",108).
інфо("Інна","Мирна","несприятливі природні умови",36).
інфо("Інна","Мирна","працевлаштування",80).
інфо("Ігор","Сірий","працевлаштування",60).

список:-write("Список 1-й: Студент чоловічої статі, старшокурсник,
                місце проживання не важливо \t"),nl,
                студент(Прізвище,Імя,чол,_,5),
                поваж_працевл(Причина,5),
                write(Прізвище,"\t",Імя,"\t",Причина),
                nl,
                fail.

список:- write("Список 2-й: Студентка, стать жін., місцева,
                курс не має значення,неповажні причини \t"),nl,
                студент(Прізвище,Імя,жін,м,_),
                причина(Причина, непов),
                write(Прізвище,"\t",Імя,"\t",Причина),
                nl,
                fail.

список:- write("Список 3-й:  Студент,місцевий з будь-якого курсу,
                стать жіноча, неповажна причина,
                кількість пропущених годин \t"),nl,
                студент(Прізвище,Імя,жін,м,_),
                причина(Причина, непов),
                інфо(Прізвище, Імя, Причина,Години),
                write(Прізвище,"\t",Імя,"\t", Причина,"=", Години,"год."),
                nl,
                fail.
goal
список.
```

Результати пошуку наступні:

```
Список 1-й: Студент чоловічої статі, старшокурсник, місце проживання не важливо
Іван    Горин    працевлаштування
Євген   Кокін    працевлаштування
Список 2-й: Студентка, стать жін., місцева, курс не має значення,неповажні причини
Інна    Пронь    стомленість
Інна    Пронь    погане самопочуття
```

```
Інна    Пронь    незацікавленість у предметі
Інна    Пронь    набридло
Інна    Пронь    недобрий ранок
Олена   Чайка    стомленість
Олена   Чайка    погане самопочуття
Олена   Чайка    незацікавленість у предметі
Олена   Чайка    набридло
Олена   Чайка    недобрий ранок
Список 3-й:  Студент,місцевий з будь-якого курсу, стать жіноча, неповажна причина, кількість пропущених годин
Інна    Пронь    незацікавленість у предметі=25год.
Олена   Чайка    набридло=24год.
no
```

У виключних випадках анонімні змінні також можна використовувати у фактах. Наприклад, якщо треба виразити мовою пролог такі факти як «у кожного є взуття, одежа, валіза» або «кожен приймає їжу», то їх запис виконується у такому вигляді:

```
predicates
  у_кожного_є(dword,symbol) - nondeterm (o,o)
  приймати_їжу(dword) - nondeterm (o)

clauses
  у_кожного_є(_,взуття).
  у_кожного_є(_,одежа).
  у_кожного_є(_,валіза).
  приймати_їжу(_).
goal
  у_кожного_є(_,Що).
```

Пролог на запит *у_кожного_є(_,Що)* («кожен володіє чимось») виводить список

```
Що=взуття
Що=одежа
Що=валіза
3 Solutions
```

2.2.8. Ініціалізація змінних

Пролог не має оператора привласнення. Не можна зберегти інформацію, привласнивши значення змінній. Це важлива відмінність Прологу від інших мов програмування.

Змінні у пролог-програмах ініціалізуються при зіставленні з константами у фактах або правилах. До ініціалізації змінна вільна, після привласнення їй значення – стає зв’язаною. Змінна залишається зв’язаною тільки той час, який необхідний для отримання рішення за запитом, потім Пролог звільняє її і шукає

інше рішення. Змінні використовуються як частина процесу пошуку рішення, а не як сховище інформації.

У наступному прикладі розглянута ілюстрація того, як і коли змінні набувають свого значення.

```
Predicates
 подобається(symbol,symbol).

clauses
 подобається (ірина, читання).
 подобається(іван,"ком'ютер").
 подобається(іван,футбол).
 подобається(сергій,футбол).
 подобається(ігор,плавання).
 подобається(ігор,читання).

goal
 подобається(Особа,читання),подобається(Особа,плавання).
```

Розглянемо запит: чи є людина, якій подобається і читання, і плавання?

Пролог вирішуватиме обидві частини запиту за допомогою пошуку фактів з початку і до кінця програми. У першій частині запиту

подобається(Особа,читання)

змінна *Особа* є вільною і її значення невідомо перед тим, як Пролог спробує знайти рішення. З іншого боку, другий аргумент, *читання*, відомий. Пролог шукає факт, який відповідає першій частині запиту. Перший факт у програмі

подобається (ірина, читання).

задовольняє першій частині запиту (*читання* у факті відповідає *читання* у запиті), Пролог пов’язує вільну змінну *Особа* із значенням *ірина*, що відповідає значенню у факті. В той же час Пролог поміщає покажчик в список фактів, що показує, як далеко просунулася процедура пошуку.

Далі, для повного дозволу запиту (пошук людини, якій подобається і читання, і плавання) повинна бути вирішена друга частина запиту. Оскільки *Особа* зараз пов’язана із значенням *ірина*, Пролог повинен шукати факт

подобається(ірина,плавання).

Пролог шукає цей факт від початку програми, але збігів немає, тому що в програмі такий факт відсутній. Друга частина запиту помилкова, якщо *Особа* має значення *ірина*.

Тепер Пролог звільнить змінну *Особа* і спробує знайти інше рішення першої частини запиту. Пошук іншого факту, що задовольняє першій частині запиту, розпочинається з покажчика в списку фактів (таке повернення до відміченої позиції називається *пошуком з поверненням*).

Пролог шукає наступну людину, яка любить читання і знаходить факт

подобається(ігор,читання).

Змінна *Особа* зараз пов'язана із значенням *ігор*, і Пролог намагається знов знайти відповідність з другою частиною запиту за допомогою пошуку факту у програмі:

подобається(ігор,плавання).

Пролог знаходить збіг (факт в програмі), і запит повністю задовольняється. Пролог (Test Goal) повертає одне рішення, що такою особою є

```
Особа=ігор
1 Solution
```

2.2.9. Зіставлення в Пролог

Пролог:

- зіставляє питання і відповіді;
- шукає зіставлення;
- зіставляє умови з фактами;
- зіставляє змінні з константами.

Зіставлення (matching) – це порівняння. У Пролозі є декілька прикладів зіставлення одного з іншим.

Ідентичні структури можуть бути зіставленими одна з одною. Наприклад, запит *подобається(ірина,читання)* зіставлено у програмі з фактом *подобається(ірина,читання)*, що підтверджується Прологом (*yes*):

```
Predicates
 подобається(symbol,symbol)

clauses
 подобається (ірина, читання).
 подобається(іван,"ком'ютер").
 подобається(іван,футбол).
 подобається(сергій,футбол).
 подобається(ігор,плавання).
 подобається(ігор,читання).
```

```
goal
 подобається(ірина,читання).
```

Звичайно зіставлення використовує одну або декілька вільних змінних. Наприклад, якщо змінна *Заняття* вільна, то *подобається(ірина,Заняття)* зіставлено з *подобається(ірина, читання)*

```
Predicates
 подобається(symbol,symbol)

clauses
 подобається (ірина, читання).
 подобається(іван,"ком'ютер")
 подобається(іван,футбол).
 подобається(сергій,футбол).
 подобається(ігор,плавання).
 подобається(ігор,читання).

goal
 подобається(ірина,Заняття).
```

і змінна *Заняття* приймає значення читання.

Якщо ж у програмі

```
predicates
 подобається(symbol,symbol) - nondeterm(o,i),(i,i)

clauses
 подобається (ірина, читання).
 подобається(іван,"ком'ютер").
 подобається(іван,футбол).
 подобається(сергій,футбол).
 подобається(ігор,плавання).
 подобається(ігор,читання).

goal
 подобається(Хто,футбол),подобається(Хто,"ком'ютер").
```

під час вирішення першої підцілі *подобається(Хто,футбол)*, змінна *Хто* зв’язується з аргументом *іван*, то під час узгодження другої підцілі змінна *Хто* вже зв’язана з *іван* і вона діє так само, як звичайна константа. Підціль *подобається(Хто,"ком'ютер")* зіставлена з *подобається(іван,"ком'ютер")*, але не зіставлена з будь-яким іншим фактом, наприклад *подобається(сергій,футбол)*. У даному прикладі вже у другій підцілі зіставлення не виконується, оскільки якщо змінна стає зв’язаною, то її значення не може змінюватися (слід звернути увагу на оголошення предиката *подобається(symbol,symbol)-nondeterm(o,i),(i,i)*, де

перший аргумент має спочатку вихідний параметр, а після зіставлення у першій підцілі – вхідний).

Як може змінна виявитися зв'язаною при спробі Прологу зіставити її з чим-небудь? Змінні не можуть зберігати значення, оскільки вони стають зв'язаними тільки на проміжок часу, необхідний для пошуку (або спроби пошуку) одного рішення цілі. Тому є тільки одна можливість для змінної виявитися зв'язаною –перед спробою зіставлення, якщо ціль вимагає більше одного кроку, і змінна стала зв'язаною на попередньому кроці. Якщо для підцілі немає рішень, Пролог «звільнить» змінну і повернеться назад, намагаючись знайти нове рішення для цілі, а потім перевірить, чи «працюватиме» ціль з новим значенням змінної.

Дві вільні змінні можуть зіставлятися одна з одною.

```
Predicates
  подобається(symbol,symbol)-nondeterm(o,i),(i,i)
  подобається(symbol)-nondeterm(o)

clauses
  подобається (ірина, читання).
  подобається(іван,"ком'ютер").
  подобається(іван,футбол).
  подобається(сергій,футбол).
  подобається(ігор,плавання).
  подобається(ігор,читання).
  подобається(Хто):-
    подобається(Хто,футбол),подобається(Хто,"ком'ютер").

goal
  подобається(Особа).
```

Результатом пошуку є одне рішення *Особа=іван*. У програмі ціль *подобається(Особа)* зіставляється з головою правила *подобається(Хто)*, зв'язуючи при цьому змінні *Особа* і *Хто* між собою, з моменту «скріплення» вони трактуються як одна змінна, і будь-яка зміна значення однієї з них приводить до негайної відповідної зміни іншої. У разі подібного «скріплення» між собою декількох вільних змінних всі вони називаються суміщеними вільними змінними. Деякі методи програмування спеціально використовують «взаємозв'язок» вільних змінних, що є, насправді, різними.

Якщо у прикладі залишаючи незмінним запит, переписати правило у вигляді

подобається(Хто):-подобається(Хто,футбол),подобається(Х,"ком'ютер").

де першому аргументу у підцілях присвоєно різні імена змінних, то Пролог буде шукати альтернативу, порівнюючи з іншими фактами у розділі фраз і виведе два рішення

```
Особа=іван
Особа=сергій
2 Solutions
```

Приклад із застосуванням змінних.

```
predicates
студент(string, string, string) - nondeterm(o,o,o)
група(string, string) -nondeterm(o,o)
ідентифікація(string, integer, integer) -nondeterm(i,i,o)
правило(string, string, integer) -nondeterm(o,o,o)

clauses
студент("Черноморець", "чоловіча", "українець").
студент("Глеб", "чоловіча", "українець").
студент("Стеба", "жіноча", "українець").
студент("Приходько", "чоловіча", "українець").
студент("Зимогляд", "чоловіча", "українець").
студент("Пушкін", "чоловіча", "росіянин").
студент("Васькін", "чоловіча", "росіянин").

група("Черноморець", "П-09-51").
група("Глеб", "П-09-51").
група("Стеба", "П-09-51").
група("Приходько", "П-09-51").
група("Зимогляд", "П-09-51").
група("Пушкін", "П-08-51").
група("Васькін", "П-07-51").

ідентифікація("Черноморець", 20, 086).
ідентифікація("Глеб", 13, 084).
ідентифікація("Стеба", 8, 089).
ідентифікація("Приходько", 9, 066).
ідентифікація("Зимогляд", 7, 089).
ідентифікація("Пушкін", 7, 065).
ідентифікація("Васькін", 13, 063).

%вивести всіх студентів та їх шифр у яких номер по списку 13
правило(X,Y,Z):-група(X,Y),ідентифікація(X,13,Z).

goal
правило(Прізвище, Шифр_групи, Зал_книжка).
/*
```

```
Прізвище=Глеб, Шифр_групи=П-09-51, Зал_книжка=84
Прізвище=Васькін, Шифр_групи=П-07-51, Зал_книжка=63
2 Solutions
*/
```

2.2.10. Коментарі

Хорошим стилем програмування є включення в програму коментарів, що пояснюють все те, що може бути незрозуміло через деякий час або іншому користувачу. Якщо підібрати відповідні імена для змінних, предикатів і доменів, то знадобиться менше коментарів, оскільки програма пояснюватиме себе «сама».

Багаторядкові коментарі повинні починатися з символів /* (коса межа та зірочка) і завершуватися символами */ (зірочка та коса межа). Для однорядкових коментарів можна використовувати ці ж символи, або починати коментар символом відсотка (%).

```
/*  Це перший коментар */
%  Це другий коментар
/***********************************************
А ці три рядки — приклад багаторядкового коментаря
***********************************************/
/*Можна помістити коментар Visual prolog /*всередині коментаря */ як вказано у цих рядках*/
```

У Visual Prolog можна використовувати коментар після кожного субдомена в оголошенні доменів:

```
domains
стаття = книга(string Назва, string Автор); автомобіль(string Марка)
```

і в оголошеннях предикатів:

```
predicates
конвертація(string Прописний, string Рядковий)
```

Слова *Назва*, *Автор*, *Марка*, *Прописний* і *Рядковий* будуть проігноровані компілятором, але зроблять програму більш читабельною.

2.3. Розділи програм Visual Prolog

Синтаксис Visual Prolog розроблений для того, щоб відображати знання про властивості і відношення об’єктів.

На відміну від інших версій Прологу, Visual Prolog – це компілятор, контролюючий типи: для кожного предиката оголошуються типи об’єктів, які

він може використовувати. Оголошення типів дозволяє програмам Visual Prolog бути скомпільованими безпосередньо в машинні коди, при цьому, швидкість виконання порівняна, а в деяких випадках і перевищує швидкість аналогічних програм на мовах C, C++.

Існує чотири основні розділи програм на Visual Prolog, де оголошуються і описуються предикати та типи аргументів, задаються правила і визначається ціль програми.

Слід детальніше розглянути синтаксис правил і оголошень і стисло розглянути інші розділи програм: бази даних, константи, різні глобальні розділи і директиви компілятора.

Звичайно програма на Visual Prolog складається з чотирьох основних програмних розділів. До них відносяться:

- розділ *clauses* (фраз);
- розділ *predicates* (предикатів);
- розділ *domains* (доменів);
- розділ *goal* (цілей);
- розділ *facts* (фактів);
- розділ *constants* (фактів);
- розділи *global* (глобальні).

Розділ *clauses* – серце Visual Prolog-програми, де записуються факти і правила, якими оперуватиме Visual Prolog, намагаючись вирішити ціль програми.

Розділ *predicates* – розділ, у якому оголошуються предикати і домени (типи) їх аргументів.

Розділ *domains* – розділ, що служить для оголошення всіх доменів користувача, що не є стандартними доменами Visual Prolog.

Розділ *goal* – розділ, в якому розміщена ціль Visual Prolog-програми.

Розділ *facts* – розділ, де оголошуються факти, що включаються в динамічну базу даних.

Розділ *constants* – розділ, де оголошуються символічні константи.

Visual Prolog дозволяє оголошувати деякі розділи domains, predicates, clauses глобальними (а не локальними), оголосивши у програмі спеціальні розділи *global domains*, *global predicates* і *global facts*.

2.3.1. Розділ фраз

У розділ *clauses* (фраз) слід помістити всі факти і правила, що складають програму, яка розробляється. Всі фрази для кожного конкретного предиката в розділі *clauses* повинні розташовуватися разом. Послідовність фраз, що описують один предикат, називається *процедурою*.

Намагаючись вирішити ціль, Visual Prolog, починаючи з першої фрази розділу *clauses*, проглядатиме кожен факт і кожне правило, прагнучи знайти зіставлення. У міру просування зверху вниз по розділу clauses, він встановлює внутрішній *покажчик* на першу фразу, що є частиною шляху, який веде до рішення. Якщо наступна фраза не є частиною цього логічного шляху, то Visual Prolog повертається до встановленого покажчика і шукає чергове відповідне зіставлення, переміщаючи покажчик на нього (цей процес називається пошук з поверненням).

2.3.2. Розділ предикатів

Visual Prolog поставляється з великим набором вбудованих предикатів.

Predefined Predicates – їх не потрібно оголошувати, а інтерактивне довідкове керівництво надає повний їх опис. Таких, наприклад, як *write*, *nl*, *fail*, cut, *not* та ін.

Якщо в розділі *clauses* програми на Visual Prolog був описаний власний предикат (створений користувачем), то слід обов'язково оголосити його в розділі *predicates* (предикатів). Інакше Visual Prolog не зрозуміє, про що ведеться мова. В результаті оголошення предиката повідомляється, до яких *доменів* (*типів*) належать аргументи цього предиката. Стандартні домени оголошувати не обов'язково.

Предикати задають факти і правила. У розділі *predicates* всі предикати просто перераховуються з вказівкою типів (доменів) їх аргументів.

Ефективність роботи Visual Prolog значно зростає саме через те, що оголошуються типи об'єктів (аргументів), з якими працюють факти і правила.

Як оголосити призначений для користувача предикат? Оголошення предиката починається з імені цього предиката, за яким йде відкриваюча (ліва) кругла дужка, після чого слід вказати нуль або більше доменів (типів) аргументів предиката. Синтаксис запису оголошення предикатів наступний:

ІмʼяПредиката(тип1 ІмʼяАргумента1, ..., типN ІмʼяАргументаN)

Після кожного домена (типу) аргумента ставиться кома, а після останнього типу аргумента – закриваюча (права) дужка. На відміну від фраз в розділі clauses, декларація предиката не завершується крапкою.

Аргументи предикатів повинні належати доменам(типам), відомим Visual Prolog. Доменами (типами) аргументів предиката можуть бути або стандартні домени, або домени, оголошені користувачем в розділі *domains*.

Існують наступні визначення типів даних, які приміняються як до даних зі знаком, так і до даних без знаку, як наведено у табл. 2.4 (дані взяті з оперативної довідкової ситеми та представляють єдину модель даних Windows).

Таблиця 2.4. Основні стандартні домени

Домен	Опис	Реалізація
1	2	3
short	Коротке, знакове, кількісне	Платформи 16 біт (-32 768 – 32 767)
ushort	Коротке, беззнакове,	Платформи 16 біт (0 – 65 535)
long	Довге, знакове, кількісне	Платформи 32 біт (-2 147 483 648 –2 147 483 647) Платформи 64-біт (-9223372036854775808 до 9223372036854775807)
ulong	Довге, беззнакове, кількісне	Платформи 32 біт (0 – 4 294 967 295) Платформи 64 біт (0 – 18446744073709551615)

Продовження таблиці 2.4

1	2	3
integer	Знакове, кількісне, має платформо-залежний розмір	Платформи 32 біт (-2 147 483 648 – 2 147 483 647) Платформи 64 біт (-9223372036854775808 – 9223372036854775807)
unsigned	Беззнакове, кількісне, має платформо-залежний розмір	Платформи 32 біт (0 – 4 294 967 295) Платформи 64 біт (0 – 18 446 744 073 709 551 615)
yte		Платформи 8 біт (0 – 55)
word		Платформи 16 біт (0 – 65 535)
dword		Платформи 32 біт (0 – 4 294 967 295) Платформи 64 біт (0 – 18446744073709551615)

Можна вказувати імена аргументів *Ім'яАргументаN* як коментар домена, що покращує читання програми і не позначається на швидкості її виконання, оскільки компілятор їх ігнорує.

Синтаксично значення, що належить одному з цілочисельних доменів, записується як послідовність цифр зі знаком мінус «-». Є також вісімкові і шістнадцятирічні синтаксиси для основних доменів.

Інші базові домени приведені у табл. 2.5.

Таблиця 2.5. Основні стандартні домени

Домен	Опис і реалізація
1	2
char	Символ, що реалізовується як беззнаковий byte. Синтаксично це символ, поміщений між двома одинарними лапками: ' a'
symbol	Послідовність символів, що реалізуються як покажчик на вхід в таблиці ідентифікаторів, яка зберігає рядки ідентифікаторів. Синтаксис аналогічний рядкам

Продовження таблиці 2.5

real	Число з плаваючою комою, що реалізовуються як 8 байт відповідно до угоди IEEE; еквівалентний типу double в *C*. Синтаксично числа з необов'язковим знаком (+ або -), за яким слідує декілька цифр *XXXXXX*, потім необов'язкова десяткова крапка (.) і ще цифри *XXXXXX*, за якими – необов'язкова експоненціальна частина. Приклади дійсних чисел (*real*): 42705.9999.86.72 або 9111.929437521e238...79.83e+21 79.83e+21 означає 79.83x1021, як і в інших мовах. Допустимий діапазон чисел: від 1x10-307 до 1x10+308 (від 1e-307 до 1e+308). При необхідності, цілі автоматично перетворяться в real.
string	Послідовність символів, що реалізовуються як покажчик на байтовий масив, що завершується нулем, як в мові *C*. Для рядків допускається два формати: 1) послідовність літер, цифр і символів підкреслення, причому перший символ повинен бути рядковою буквою. 2) послідовність символів, поміщених в подвійні лапки. Приклади рядків: *логічне_програмування* або *" Логічне програмування "* Рядки у пролог-програмі можуть досягати довжини в 255 символів, тоді як рядки, які система Visual Prolog прочитує з файлу або будує усередині себе, можуть досягати (теоретично) до 4 Гбайт на 32-бітових платформах

Домени типів *byte*, *word* і *dword* найбільш зручні при роботі з машинними числами. В основному використовуються типи *integer* і *unsigned*, а також *short* і *long* (і їх беззнакові аналоги) для більш спеціалізованих додатків.

У оголошеннях доменів ключові слова *signed* і *unsigned* можуть використовуватися разом із стандартними доменами типів *byte*, *word* і *dword* для побудови нових базових доменів.

Так:

```
domains
i8 = signed byte
```

створює новий базовий домен в діапазоні від -128 до +127.

У тих випадках, коли необхідно встановити дані чітко визначеного розміру, у Visual Prolog 7.4 передбачені додаткові типи даних. Позначення типів даних фіксованої точності отримують зі звичайних позначень типів даних Win32, таких як *dword* або *long*, шляхом додавання суфіксу розміру:

- dword32 (32-бітове ціле без знаку);
- dword64 (64- бітове ціле без знаку);
- integer32 (32- бітове ціле зі знаком);
- integer64 (64- бітове ціле зі знаком);
- long32 (32- бітове довге ціле зі знаком);
- long64 (64 - довге довге ціле зі знаком);
- unsigned32 (32- бітове кількісне без знаку);
- unsigned64 (64- бітове кількісне без знаку);
- ulong32 (ціле типу long32 без знаку);
- ulong64 (ціле типу long64 без знаку).

Точність таких типів даних відображає зміну точності покажчиків, тобто, вони стають 32-бітовими у коді Win32 та 64-бітовими у коді Win64. Таким чином, ці типи даних забезпечують автоматичну зміну розмірів цілочисельних типів даних залежно від зміни розмірів покажчиків, у зв’язку з чим їх іноді називають поліморфними.

До основних нових функцій Visual Prolog 7.4 відноситься здатність генерувати 64-розрядні програми (тільки для Commercial Edition), що дозволяє писати програми, які потребують більше ресурсів комп’ютера (для забезпечення сумісності вхідного коду між 32 і 64 бітними програмами декілька типів були змінені). Використання 64-розрядних ОС істотно в тих випадках, коли Windows відводиться помітна роль у прикладних корпоративних та професійних системах.

У Visual Prolog версії 7.4 при налаштуванні проекту є можливість вибору платформи для:

- 32- розрядні;
- 64-бітні;
- 32 біт + 64 біт проектів.

За замовчуванням встановлено параметр 32 біта для старих/існуючих проектів та за замовчуванням використовується платформа 32 біт + 64 біт для нових проектів.

Основною ціллю візуальної підтримки 64-бітних платформ Prolog является те, що вона повинна мати можливість створювати як 32-разрядні та 64-бітні програми з одного вхідного проекту, без умовної компіляції, за виключенням дуже рідких випадків. Тому PFC (Prolog Foundation Classes – базові класи мови Prolog) був оновлений, щоб підтримувати обидві платформи з єдиною базою даних.

Ідентифікатори (*symbol*) і *рядки* (*string*) взаємозамінні в програмі, проте Visual Prolog зберігає їх роздільно.

Ідентифікатори зберігаються в *таблиці ідентифікаторів*, а для уявлення використовуються лише їх індекси в цій таблиці, але не самі рядки ідентифікаторів. Це означає, що зіставлення ідентифікаторів виконується дуже швидко, а у випадку якщо вони зустрічаються в програмі кілька разів, то і зберігання їх компактне.

Рядки ж не зберігаються в пошуковій таблиці, і при необхідності зіставлення Visual Prolog перевіряє їх символ за символом. Користувач сам повинен визначати, який домен краще використовувати в кожній конкретній програмі. Приклади простих об’єктів, що належать до основних стандартних доменів наведені у таблиці 2.6.

Таблиця 2.6. Основні стандартні домени

Простий об’єкт	Домен
іван, «Іван», Жовті_Води, «Жовті Води»	symbol або string
-1, 7, 9, 0, 100	Integer
0.15, 25.45, -15.5, 245.4e+7	Real
'a', 'A', '/', '&'	Char

Якщо предикат *мій_предикат(symbol, integer)* оголошений в розділі *predicates* таким чином:

```
predicates
мій_предикат(symbol, integer)
```

то не потрібно в розділі *domains* декларувати домени його аргументів, оскільки *symbol* і *integer* є стандартні домени. Проте, якщо цей же предикат оголосити доменами користувача:

```
predicates
мій_предикат (назва, номер)
```

то необхідно оголосити у розділі *domains* , що назва – це символічний тип, номер – це цілий тип, та обидва належать до стандартних доменів *symbol* і *integer*:

```
domains
назва = symbol
номер = integer
predicates
мій_предикат (назва, номер)
```

Або, припустимо, потрібно описати предикат, який відображає позицію символу в алфавіті

```
predicates
 поз_симв_алф (char, integer)   % або алф_поз (char,   unsigned)

clauses
 поз_симв_алф ('а', 1).
 поз_симв_алф ('б', 2).
 %    тут знаходиться решта літер
 поз_симв_алф ('я', 32).

goal
%декілька простих цілей, які можна поставити до фактів

поз_симв_алф ('а',1).                % відповідь: Yes
% поз_симв_алф (Х, 3).               % відповідь: Х=в
% поз_симв_алф ('я',Позиція).        % відповідь: Позиція=32
```

Для одного предиката (наприклад список) допускається декілька оголошень:

```
predicates
список(імена, імена_ у_списку)
список(номер,номер_у_списку),
```

де домени в даному випадку визначені користувачем.

Зовсім не обов’язково, щоб при зіставленні двох Visual Prolog-змінних вони належали одному і тому ж домену. Змінні можуть бути пов’язані з константами з різних доменів. Таке вибіркове змішування допускається,

оскільки Visual Prolog *автоматично виконує перетворення типів* (з одного домена в інший), але тільки в наступних випадках:

- між *рядками* (string) і *ідентифікаторами* (symbol);
- між *цілими* (integer), *дійсними* (single) і *символами* (char).

При перетворенні символу в числове значення цим значенням є величина ASCII коду символу.

Аргумент з домена *мій_дом*, який оголошений таким чином:

```
domains
мій_дом = <базовий domain>    % <базовий domain>
```

базовий domain – це стандартний домен, який може вільно змішуватися з аргументами з цього основного домена і з аргументами всіх сумісних з ним стандартних доменів.

Якщо основний домен – *string*, то з ним сумісні аргументи з домена *symbol*; якщо ж основний домен *integer*, то з ним сумісні домени *real*, *char*, *word* і ін. Таке перетворення типів, наприклад, означає, що можна:

- викликати предикат з аргументами типу *string*, задаючи йому аргументи типу *symbol*, і навпаки;
- передавати предикату з аргументами типу *real* параметри типу *integer*;
- передавати предикату з аргументами типу *char* параметри типу *integer*;
- використовувати у виразах і порівняннях символи без необхідності отримання їх ASCII кодів.

Існує набір правил, що визначають, до якого домена належить результат змішування різних доменів.

Можна оголошувати предикати з різною *арністю* – кількість аргументів, які він містить. Факти програми можуть містити два предикати з одним і тим же ім’ям, але відрізнятися арністю.

У розділах *predicates* і *clauses* версії предикатів з одним ім’ям і різною арністю повинні розміщуватися разом, за винятком обмеження – різна арність завжди визначається як повна відмінність предикатів.

2.3.3. Розділ доменів

У традиційному Пролозі є тільки один тип – *терм*. У Visual Prolog оголошують домени всіх аргументів предикатів. Домени дозволяють задавати різні імена різним типам даних, які, інакше, виглядатимуть абсолютно однаково.

У програмах Visual Prolog об'єкти у відношеннях/властивостях (аргументи предикатів) належать доменам, причому це можуть бути як стандартні, так і описані користувачем спеціальні домени.

Розділ *domains* служить двом корисним цілям.

По-перше, можна задати доменам осмислені імена, навіть якщо внутрішньо ці домени аналогічні вже наявним стандартним.

По-друге, оголошення спеціальних доменів використовується для опису структур даних, відсутніх в стандартних доменах.

Іноді дуже корисно описати новий домен – особливо, коли потрібно пояснити окремі частини розділу *predicates*. Оголошення власних доменів, завдяки привласненню осмислених імен типам аргументів, допомагає документувати описані користувачем предикати.

Наступний приклад показує як оголошення доменів допомагає документувати предикати.

У розділі пролог-програми збережені факти про студентів (прізвище, шифр групи, три останніх цифри залікової книжки та порядковий номер у списку групи). Можна оголосити відповідні предикати, використовуючи стандартні домени:

```
predicates
  прізвище(string,string,word) - nondeterm(o,i,o)
  список(word, string) - nondeterm(o,o)

clauses
  прізвище("Бурлаченко","П-08-51",100).
  прізвище("Бачанцева","П-08-51",101).
  прізвище("Барчук","П-08-51",102).
  прізвище("Біла","П-08-51",103).
  прізвище("Богдан","П-08-51",104).
  список(1,"Бурлаченко").
  список(2,"Бачанцева").
```

```
 список(3,"Барчук").
 список(4,"Біла").
 список(5,"Богдан").

goal
прізвище(X,"П-08-51",Y).
```

Перевіривши всі факти для цілі «знайти прізвища студентів групи П-08-51 та три останні цифри їх залікових книжок», Пролог напише

```
X=Бачанцева, Y=101
X=Барчук, Y=102
X=Богдан, Y=104
3 Solutions
```

В більшості випадків таке оголошення працюватиме дуже добре. Проте припустимо, що декілька місяців опісля написання програми, її вирішили змінити. Є побоювання, що подібне оголошення цього предиката абсолютно нічого не скаже програмісту про об’єкти, що використовує предикат у якості аргументів. І напроти, декларація цього ж предиката, представлена нижче, допоможе розібратися в тому, що ж є аргументи даного предиката:

```
domains
 студент, шифр_групи=string
 номер_списку, залік_книжка=word

predicates
 прізвище(студент,шифр_групи,залік_книжка) - nondeterm(i,i,o)
список(номер_списку, студент) - nondeterm(o,o)
```

вказавши у цілі осмислені імена змінним

```
goal
список(Номер,Студент),прізвище(Студент,"П-08-51",Залік_книжка).
```

отримаємо рішення

```
Номер=2, Студент=Бачанцева, Залік_книжка=101
Номер=3, Студент=Барчук, Залік_книжка=102
Номер=5, Студент=Богдан, Залік_книжка=104
3 Solutions
```

Однією з головних переваг оголошення власних доменів є те, що Visual Prolog може відстежувати помилки типів, наприклад у наведеному правилі змінна *X* у різних предикатах належить різним типам *string* і *word*.

```
Протокол(X,Y):- прізвище(X,_ Y), список(X,_).
```

У наступному правилі

```
Протокол(X,Y):- прізвище(X,Y,_), прізвище(Y,X,_).
```

де дивлячись на те, що прізвище і шифр групи описуються як *string*, вони не еквівалентні один одному. Це і дозволяє Visual Prolog визначити помилку, якщо їх переплутати. Це корисно в тих випадках, коли програми дуже великі і складні.

Чому ж не можливо використовувати спеціальні домени для оголошення всіх аргументів, якщо вони привносять більше сенсу в позначення аргументів? Відповідь полягає в тому, що аргументи з типами із спеціальних доменів не можуть змішуватися між собою, навіть якщо ці домени однакові.

Саме тому, не дивлячись на те, що прізвище і шифр групи належать одному домену *string*, вони не можуть змішуватися. Проте всі власні домени користувача можуть бути зіставлені стандартним доменам.

Іноді рішення приводить до помилки типу. Це може трапитися через те, що мала місце спроба передати результуюче значення предиката, яке відноситься до одного домена, як один з аргументів в інший предикат з аргументами того ж типу. Це приводить до помилки, оскільки перший домен відрізняється від другого домена. І хоча обидва ці домени відповідають одному типу – це різні домени. Тому, якщо змінна у фразі використовується більш ніж в одному предикаті, вона повинна бути однаково оголошена в кожному з них.

Як можна використовувати оголошення доменів для відстежування помилок типу

```
domains
  марка= string
  колір = symbol
  вік = byte
  вартість, спідометр = ulong

predicates
  автомобіль(марка,спідометр,вік,колір,вартість)

clauses
автомобіль ("Chrysler",130000,3,червоний,12000).
автомобіль ("Ford",90000,4,сірий,25000).
автомобіль ("Datsun",8000,1,чорний,30000).
```

Тут предикат *автомобіль*, оголошений в розділі *predicates*, має 5 аргументів. Один з них відноситься до домена *вік* типу *byte*. В сімействі

процесорів x86 тип *byte* – це 8-бітове беззнакове ціле, яке може приймати значення від 0 до 255, включаючи межі. Аналогічно, домени *спідометр* і *вартість* типа *ulong*, який є 32-бітовими беззнаковими цілими, домен *марка* є типу *string* і домен *колір* – символьного типу (*symbol*).

Кожна з наступних цілей:

```
автомобіль ("Chrysler",13,40000,червоний,12000).
автомобіль ("ford",90000,сірий,4,25000).
автомобіль (4,червоний,30000,80000,"datsun").
```

приведе до помилки розпізнавання типу. В першому випадку це відбудеться через те, що *вік* повинен бути типу *byte*. Отже, *Visual Prolog* зможе легко визначити, що при введенні цієї цілі об'єкти *спідометр* і *вік* в предикаті *автомобіль* були переплутані місцями. В другому випадку були переплутані *вік* і *колір*.

Ще один приклад застосування власних доменів користувача, які виконують також роль коментарів.

```
domains
номер, надійшло, реалізовано, ціна = integer

predicates
run()
магазин(номер, надійшло, реалізовано, ціна)-nondeterm(o,o,o,o)

clauses
магазин(37, 38, 19, 301).
магазин(18, 45, 27, 311).

run:-магазин(Номер, Поступило, Реалізовано, Ціна),
write("Магазин №", Номер),
Залишок_сума = (Поступило *Ціна - Реалізовано *Ціна),
Залишок_товару = (Поступило - Реалізовано),
write("залишок=", Залишок_товару" на сумму=", Залишок_сума," грн."),
nl,
fail.
run.

goal
run.

/*Магазин №37  залишок=19 на сумму=5719 грн.
Магазин №18  залишок=18 на сумму=5598 грн.
yes*/
```

2.3.4. Розділ цілі

По суті, пролог-програма може складатися лише з одного розділу *goal*:

```
goal
write("Програму виконано \n").
```

Результатом є виведений текст та підтвердження успішного виконання цілі:

```
Програму виконано
yes
```

Цілі аналогічні тілу правила – це просто список підцілей. Ціль відрізняється від правила лише тим, що за ключовим словом goal не слідують символи «:-» (так, якщо).

```
predicates
  проживання(symbol,symbol,symbol) - nondeterm (o,i,i)
  внз(symbol,symbol,symbol) - nondeterm (i,i,o)
  секція_конференції(symbol,symbol,symbol,symbol) - nondeterm (o,o,o,o)

clauses
  проживання("Бойко", жін, так).
  проживання("Бутко", чол, ні).
  проживання("Вороніна", жін, так).
  проживання("Кубська", чол, ні).
  проживання("Ротан", чол, так).
  проживання("Федір", жін, ні).

внз("Бойко", "Дніпропетровськ", "ДМА").
внз("Бутко", "Маріуполь", "МПУ").
внз("Вороніна", "Жовті Води", "ІП Стратегія").
внз("Кубська", "Кривий Ріг", "КЕУ").
внз("Ротан", "Дніпропетровськ", "ДНУ").
внз("Федір", "Київ", "КНУ").

секція_конференції("Бойко", "Математика і ІТ", тези, ні).
секція_конференції("Бутко", "Проблеми управління інформацією",
                    доповідь, так).
секція_конференції("Вороніна","БІ у сфері середнього бізнесу", тези, так).
секція_конференції("Кубська","Проблеми ІТ-освіти", доповідь, ні).
секція_конференції("Ротан","Аналіз поширення новітніх ІТ", доповідь, так).
секція_конференції("Федір","Розробка модулів прикладних рішень ІСУ ВПП",
                    тези, ні).

/*знайти прізвища учасників конференції та назву їх статей, якщо студенти з різних вузів м.Дніпропетровська*/

goal
секція_конференції(Учасник, Стаття, _, _),
                    внз(Учасник, "Дніпропетровськ", ВНЗ).
```

Результатом пошуку є список:

```
Учасник=Бойко, Стаття=Математика і ІТ, ВНЗ=ДМА
Учасник=Ротан, Стаття=аналіз поширення новітніх ІТ у сучасному суспільстві, ВНЗ=ДНУ
2 Solutions
```

При запуску програми Visual Prolog виконує ціль автоматично. Це відбувається так, як ніби Visual Prolog викликає goal, запускаючи тим самим програму, яка намагається вирішити тіло правила goal. Якщо всі підцілі в розділі *goal* істинні,

```
predicates
  книга(symbol,symbol) - nondeterm (i,i)

clauses
  книга("Вовчок","Современник").
  книга("Гоголь","Вий").
  книга("Шевченко","Княжна").
  книга("Украинка","Боярыня").
  книга("Донцова","Крутые наследнички").
  книга("Лермонтов","Мцыри").
  книга("Достоевский","Идиот").
  книга("Пушкин","Дубровский").
  книга("Шекспир","Гамлет").

goal
книга("Шекспир","Гамлет").
```

програма завершується успішно – *yes*.

Якщо ж якась підціль з розділу *goal* помилкова, то вважається, що програма завершується неуспішно (хоча чисто зовні ніякої різниці в цих випадках немає, програма просто завершить свою роботу).

Для цілі

```
goal
книга("Пушкин","Евгений Онегин").
```

програма завершується неуспішно (*no*), тому що даний факт відсутній.

2.3.5. Розділ фактів

Програма на Visual Prolog є набором фактів і правил. Іноді в процесі роботи програми буває необхідно модифікувати (змінити, видалити або додати) деякі з фактів, з якими вона працює. В цьому випадку факти розглядаються як *динамічна* або *внутрішня* база даних, яка при виконанні програми може змінюватися. Для оголошення фактів програми, що розглядаються як частини динамічної бази даних, Visual Prolog включає спеціальний розділ – *facts*.

Ключове слово *facts* оголошує розділ фактів. Саме в цій секції Ви оголошуєте факти, що включаються в динамічну базу даних. У ранніх версіях Visual Prolog для оголошення розділу фактів використовувалося ключове слово *database*, тобто ключове слово *facts* – синонім застарілого ключового слова *database*. У Visual Prolog є декілька вбудованих предикатів, що полегшують використання динамічних фактів, як наведено у наступному коді програми.

```
facts
  xpositive(symbol,symbol)
  xnegative(symbol,symbol)

predicates
  назва_фільму(symbol) - nondeterm (o)
  стать_режисера(symbol) - nondeterm (i)
  інфо(symbol) - nondeterm (i)
  ask(symbol,symbol,symbol) - determ (i,i,i)
  remember(symbol,symbol,symbol) - determ (i,i,i)
  positive(symbol,symbol) - determ (i,i)
  negative(symbol,symbol) - determ (i,i)
  clear_facts() - nondeterm ()
  run() - nondeterm ()

clauses
  назва_фільму(чого_хочуть_жінки):-
  стать_режисера(режисер),
  інфо(жанр),
  інфо(роль),
  інфо(трид),
  інфо(книга),
  positive("Режисером є",жінка),
  positive("Відноситься до жанру",комедія),
  positive("Головну роль зіграв",мел_гібсон),
  positive("Фільм в 3D",не_показували),
  positive("Фільм за книгою",не_знятий).

назва_фільму(аліса_в_країні_чудес):-
стать_режисера(режисер),
інфо(жанр),
інфо(роль),
інфо(трид),
інфо(книга),
positive("Режисером є",чоловік),
positive("Відноситься до жанру",фантастика),
positive("Головну роль зіграв",джонні_депп),
positive("Фільм в 3D",показували),
positive("Фільм за книгою",знятий).

  назва_фільму(олександр):-
  стать_режисера(режисер),
```

```
  інфо(жанр),
  інфо(роль),
  інфо(трид),
  інфо(книга),
  positive("Режисером є",чоловік),
  positive("Відноситься до жанру",історичний),
  positive("Головну роль зіграв",колін_фаррелл),
  positive("Фільм в 3D",не_показували),
  positive("Фільм за книгою",знятий).

стать_режисера(режисер):-positive("Режисером є",жінка).
стать_режисера(режисер):-positive("Режисером є",чоловік).
  інфо(жанр):-positive("Відноситься до жанру",комедія).
  інфо(жанр):-positive("Відноситься до жанру",фантастика).
  інфо(жанр):-positive("Відноситься до жанру",історичний).
  інфо(роль):-positive("Головну роль зіграв",мел_гібсон).
  інфо(роль):-positive("Головну роль зіграв",джонні_депп).
  інфо(роль):-positive("Головну роль зіграв",колін_фаррелл).
  інфо(трид):-positive("Фільм в 3D",не_показували).
  інфо(трид):-positive("Фільм в 3D",показували).
  інфо(книга):-positive("Фільм за книгою",не_знятий).
  інфо(книга):-positive("Фільм за книгою",знятий).

positive(X,Y):-
xpositive(X,Y),!.

positive(X,Y):-
not(xnegative(X,Y)),
ask(X,Y,yes).

negative(X,Y):-
xnegative(X,Y),!.

negative(X,Y):-
not(xpositive(X,Y)),
ask(X,Y,no).
ask(X,Y,yes):-
!,
write(X," ",Y,'\n'),
readln(Reply),nl,        %readln(STRING StringVariable)
frontchar(Reply,'y',_), %frontchar(STRING String, CHAR FirstChar, STRING
                         % RestString)
remember(X,Y,yes).

ask(X,Y,no):-
!,
write(X," ",Y,'\n'),
readln(Reply),nl,
frontchar(Reply,'n',_),
remember(X,Y,no).

remember(X,Y,yes):-              % зберігає дані у внутрішній БД
```

```
    assertz(xpositive(X,Y)).        % синтаксис    assertz(<facts_domain> Fact)

remember(X,Y,no):-
    assertz(xnegative(X,Y)).

clear_facts:-
write("\n\n Натисніть Enter для завершення програми \n"),
retract(_,dbasedom),      % уніфікує та видаляє кожен факт
readchar(_).              %  внутрішньої бази dbasedom

run:-
назва_фільму(X),!,
write("\nМожливо це фільм",X),
nl,
clear_facts.

run:-
write("\nНе в змозі визначити"),
write("який це фільм.\n\n"),
clear_facts.

goal
run.
```

2.3.6. Розділ констант

Для оголошення і використання у пролог-програмах символічних констант створюють розділ *constants*. Ідентифікатори констант є глобальними і можуть оголошуватися тільки один раз. Множинне оголошення одного і того ж ідентифікатора приведе до повідомлення про помилку «Constant identifier can only be declared once» (Ідентифікатор константи може оголошуватися тільки один раз). За необхідності у програмі може бути декілька розділів *constants*, проте оголошення константи повинне проводитися перед її використанням.

Синтаксис оголошення констант:

<Id> = <Макровизначення>

де: *<Id>* – це ім’я символічної константи;

<макровизначення> – це вираз, константа чи формула, що привласнюється цій константі. Кожне *<макровизначення>* завершується символом нового рядка *(\n*), тобто на одному рядку може бути тільки один опис константи. Оголошені таким чином константи можуть пізніше використовуватися в програмах, наприклад:

```
Constants
```

```
нуль = 0
Один = 1
два = 2
сто = (10* (10-1)+10)
пі = 3.141592653

goal
write("Результат:"),A=сто*20.
```

Відповідь Пролог буде такою:

```
Результат: A=2000
1 Solution
```

У описах констант система не розрізняє верхній і нижній регістри. Отже, при використанні в розділі програми *clauses* ідентифікатора типу *constants*, його перша буква повинна бути рядковою для того, щоб уникнути плутанини між константами і змінними.

Тому наступний фрагмент програми є допустимою конструкцією:

```
Constants
Нуль = 0
Один = 1
два = 2

predicates
виконати()
clauses
виконати:-P=один+два,Q=нуль,write(P,">",Q,"\n").

goal
виконати.
```

Пролог виведе відповідь: *3>0 yes*.

Ще один приклад з використання констант:

```
predicates
run()-nondeterm()

constants
% час обробки на станках однієї деталі A і B в годинах
A_t_t=0.5
A_t_s=0.4
A_t_sh=0.3
B_t_t=0.8
B_t_s=0.6
B_t_sh=0.4
A=700 %кількість деталей A, шт.
B=500 %кількість деталей B, шт.

clauses
```

```
%Обчислити загальний час обробки деталей A і B на кожному станку
run:-write("Обробка на токарному станку\t\t= "),
S=(a*a_t_t+b*b_t_t),write(S," годин"),nl,fail.
run:-write("Обробка на свердлильному станку\t= "),
S=(a*a_t_s+b*b_t_s),write(S," годин"),nl,fail.
run:-write("Обробка на шліфувальному станку\t= "),
S=(a*a_t_sh+b*b_t_sh),write(S," годин"),nl,fail.

goal
run.
```

Рішення цієї задачі наступне:

```
Обробка на токарному станку        = 750 годин
Обробка на свердлильному станку    = 580 годин
Обробка на шліфувальному станку    = 410 годин
yes
```

Опис константи не може посилатися сам на себе, наприклад не коректний запис

```
ціна = ціна+ціна*10% або кількість=кількість - 25
```

виведе повідомлення про помилку.

2.4. Ключові слова

Програма Visual Prolog складається з коду, який перемежовується відповідними ключовими словами, які інформують компілятор коду про те, що він повинен генерувати. Наступні слова являються зарезервованими та поділяються на дві групи.

Головні:

class clauses constants constructors	implement inherits interface
delegate domains	monitor
end	namespace
facts	open
goal guard	predicates
supports	properties
resolve	

Допоміжні:

failure finally foreach from	align and anyflow as
If	Bitsize
Language	Catch
mod multi	determ digits div do
nondeterm	else elseif erroneous externally
Or	Rem
procedure	Single
Quot	then to try

Наприклад, є ключові слова, які розрізняють для предикатів і доменів оголошення від визначень. Як правило, кожен розділ розпочинається

ключовим словом. Існує правило: немає ключового слова, яке означає закінчення певної секції. Наявність іншого ключового слова вказує на закінчення попереднього розділу і початок наступного.

Виключенням з цього правила є ключові слова *Implement* (реалізація) і *end implement* (кінець реалізаціїI). Код, що міститься між цими двома ключовими словами, використовується для певного класу.

Ключове слово *open* призначене для розширення області видимості класу та використовується тільки після ключового слова *implement* (реалізація).

Ключове слово *constants* використовується для позначення розділу коду, який визначає деякі часто використовувані значення в коді програми.

Ключове слово *domains* використовується для позначки секції оголошення доменів, які будуть використовуватися в коді.

Ключове слово *facts* позначає розділ, який оголошує факти, які будуть використовуватися пізніше в коді програми. *predicates*, *clauses*, *goal*.

Ключові слова не можна використовувати як імена, визначені користувачем.

2.5. Знаки пунктуації

Знаки пунктуації для компілятора мають синтаксичний та семантичний сенс. Деякі з них можуть використовуватися у якості операторів (табл. 2.7).

Таблиця 2.7. Знаки пунктуації

Знак	Назва	Призначення
1	2	3
;	Крапка з комою	У фразах означає «АБО»
!	Знак оклику	Відсікання, видалення точок повернення
,	Кома	У фразах означає «І». У списках розділяє елементи.
.	Крапка	Завершує факт або правило.
#	Решітка	Вказує на директиву компілятора.

[ ]	Квадратні дужки	Включає список елементів, розділених комами.
( )	Круглі дужки	Включає список аргументів, розділених комами.
:-	Вертикальна двокрапка і тире	Використовується у правилах для поділу на голову і перелік дій.
:	Вертикальна двокрапка	Використовується у визначеннях предикатів.
: :	Подвійна вертикальна двокрапка	Розділяє ім'я класу та поля класу.

2.6. Директиви компілятора

Visual Prolog підтримує декілька директив компілятора, які можна додавати в програму для повідомлення компілятору спеціальних інструкцій по обробці програми при її компіляції.

Для того, щоб уникнути багаторазового набору процедур, що повторюються, можна використовувати директиву *include*:

- створити файл (*.PRO*), в якому оголосити найбільш часто використовувані предикати (за допомогою розділів *domains* і *predicates*) і дати їх опис в розділі *clauses*;
- написати початковий текст програми, яка використовуватиме ці процедури;
- у будь-яке місце початкового тексту програми (в якому можна розташувати декларацію основних розділів) розмістити рядок: *include «назва_файла.pro»;*
- під час компіляції початкових текстів програми Visual Prolog вставить зміст вказаного файлу *.PRO* (*«назва_файла.pro»*) прямо у остаточний текст поточного файлу для компіляції.

Директиву *include* можна використовувати для включення в початковий текст практично будь-якого часто використовуваного фрагмента. Крім того, будь-який файл, що включається в програму, може, у свою чергу, включати

інший файл (проте кожен файл може бути включений в програму тільки один раз).

Приклад використання *include.*

Файл «*include1.pro*»:

```
predicates
one(integer)- nondeterm(o)

Clauses
one(500).
one(700).
```

Виконати файл «*include2.pro*», який бере факти з файлу «*include1.pro*»

```
include "include1.pro"

predicates
two(integer)- nondeterm(o)

Clauses
two(100).
two(50).

goal
one(X),two(Y),Q=X+Y
```

Рішення Пролог:

```
X=500, Y=100, Q=600
X=500, Y=50, Q=550
X=700, Y=100, Q=800
X=700, Y=50, Q=750
4 Solutions
```

Для наступного прикладу створюємо файл «*include0.pro*»

```
predicates
zero(integer)- nondeterm(o)

clauses
zero(15).
zero(25).
```

Додаємо директиву *include* у файл «*include1.pro*»

```
include "include0.pro"

predicates
one(integer)- nondeterm(o)

clauses
one(500).
one(700).
```

```
goal
one(F),zero(K),V=F+K.
```

Якщо задати ціль *one(F),zero(K),V=F+K*, отримаємо

```
F=500, K=15, V=515
F=500, K=25, V=525
F=700, K=15, V=715
F=700, K=25, V=725
4 Solutions
```

Для наступного прикладу слід закоментувати ціль у файлі «*include1.pro*»

```
include "include0.pro"

predicates
one(integer)- nondeterm(o)

clauses
one(500).
one(700).

/*goal
one(F),zero(K),V=F+K.*/
```

та виконати ціль

```
goal
one(X),nl,two(Y),nl,zero(P),Q=X+Y+P.
```

з файлу «*include2.pro*».

```
include "include1.pro"

predicates
two(integer)- nondeterm(o)

clauses
two(100).
two(50).

goal
one(X),nl,two(Y),nl,zero(P),Q=X+Y+P.
```

Отримаємо відповідь Пролог

```
X=500, Y=100, P=15, Q=615
X=500, Y=100, P=25, Q=625

X=500, Y=50, P=15, Q=565
X=500, Y=50, P=25, Q=575

X=700, Y=100, P=15, Q=815
X=700, Y=100, P=25, Q=825
```

```
X=700, Y=50, P=15, Q=765
X=700, Y=50, P=25, Q=775
8 Solutions
```

Приклад використання директиви *include* для об'єктів символьного типу. Створити файл «*1.pro*», у якому присутній факт про прізвище особи

```
predicates
pred1(symbol)- nondeterm(o)

clauses
pred1("Глеб"). % прізвище,
```

та звернутися до нього у файлі «*2.pro*», який містить факти про імена родичів

```
include "1.pro"

predicates
pred2(symbol)- nondeterm(o)
run()- nondeterm()

clauses
pred2("Ольга").
pred2("Анатолій").
run:-pred1(X),nl,pred2(Y),write(X," ",Y),nl,fail.
run.

goal
run.
```

Результат пошуку *Глеб Ольга* та *Глеб Анатолій*.

Пролог-програма складається з цілого ряду одиниць компіляції. Компілятор компілює кожну з них окремо. Результатом компіляції є об'єктний файл. Такі об'єктні файли зв'язані разом для реалізації цілі проекту. Програма повинна містити рівно один розділ *goal*, який є точкою входження в програму. Одиниця компіляції повинна бути автономною у тому сенсі, що всі посилання на імена повинні бути оголошені або визначені в одиниці компіляції.

2.7. Аналіз потоку параметрів

Зв'язування змінних із значеннями у Пролог-програмах проводиться двома способами: на вході і виході. Напрям, в якому передаються значення, вказується в шаблоні *потоку параметрів* (надалі «потік параметрів»). Коли змінна передається у фразу, вона вважається вхідним аргументом і позначається символом (*i – input*), коли повертається з фрази – вихідним

аргументом і позначається символом (*o – output*). Існує чотири варіанти потоку параметрів:

(o, o) (o, i) (i, o) (i, i)

Під час компіляції програми, розпочинаючи з цільового твердження, Visual Prolog виконує глобальний аналіз потоку параметрів усієї програми. Потік параметрів у програмі визначається для кожного звернення до кожного предиката та декларується разом з оголошенням предикатів у розділі *predicates*, як наведено у наступному прикладі: потрібно вивести список прізвищ викладачів та їх навантаження (в годинах), якщо викладачі заключили договір на бюджетній основі.

```
predicates
посада(string,string) - nondeterm (i,o)
навантаження(string,integer) - nondeterm (o,o)
договір(string,string) - nondeterm (i,i)

clauses
посада("викладач","Девяткин").
посада("асистент","Ульянова").
посада("асистент","Карпов").
посада("старший викладач","Рудяков").
посада("викладач","Горовий").
посада ("викладач","Ратний").
% навантаження в годинах
навантаження("Девяткин",700).
навантаження("Ульянова",700).
навантаження("Рудяков",800).
навантаження("Ратний",850).
навантаження("Горовий",750).
навантаження("Карпов",700).
% договір
договір("Девяткин","бюджет").
договір("Ульянова","контракт").
договір("Рудяков","бюджет").
договір("Горовий","контракт").
договір("Ратний","бюджет").
договір("Карпов","контракт").
goal
навантаження(Викладач,_всього_годин),договір(Викладач,"бюджет").
```

Під час першого звернення цільового твердження до предикату *навантаження* змінні (вихідні параметри) *Викладач* та *_всього_годин* являються вільними (не зв’язаними), тому предикат *навантаження* визивається з потоком параметрів *(o,o)*.

У зверненні до предиката *договір* змінна *Викладач* є вже зв'язаною (у першій підцілі), другий аргумент цього предиката відомий, тому предикат *договір* визивається з потоком параметрів *(i,i)*.

Пролог виведе три рішення:

```
Викладач=Девяткин, _всього_годин=700
Викладач=Рудяков, _всього_годин=800
Викладач=Ратний, _всього_годин=850
3 Solutions
```

Для кожного потоку параметрів, з якими визивається предикат користувача, аналізатор потоку проходить через фрази предиката зі змінними з голови предиката, визначаючи, чи є вони вхідними чи вихідними змінними потоку, що аналізується.

Якщо додати до існуючої цілі третю підціль, що уточнює список викладачів (за посадою «викладач»)

```
goal
навантаження(Викладач,_всього_годин),договір(Викладач,"бюджет"),
посада("викладач",Викладач).
```

то знову та ж зв'язана змінна *Викладач* у другій підцілі (предикат *договір*) передається у третій предикат як вже відомий параметр (*i*), тобто предикат *посада* визивається з потоком параметрів *(i,i)*. Результатом пошуку є список:

```
Викладач=Девяткин, _всього_годин=700
Викладач=Ратний, _всього_годин=850
2 Solutions
```

У випадку, коли аналізатор потоків визначає, що предикат визивається з неіснуючим потоком параметрів, Visual Prolog виводить повідомлення про помилку (з підказкою), що допоможе полегшити визначення невірних потоків при використанні стандартних або створенні власних предикатів. Якщо, наприклад для предиката *посада* визначити параметри (*o,i*) для цілі

```
goal
навантаження(Викладач,_всього_годин),договір(Викладач,"бюджет"),
посада("викладач",Викладач).
```

і виконати ціль, то у вікні *Errors(Warnings)* з'явиться повідомлення про помилку «*E;Test_Goal, pos: 978, 703 This flow pattern doesn't exist посада(i,i)*».

Наступний приклад демонструє звертання до одного предиката двічі зі складної цілі:

```
predicates
подобається(symbol,symbol)- nondeterm (o,o),nondeterm(i,o)

clauses
подобається(ігор,ірина).
подобається(ірина,ігор).
подобається(ігор,машина).

goal
% складний запит: Кому подобається все, що подобається Ігору.
подобається(Особа,Хто),подобається(ігор,Що).
```

У прикладі потік параметрів предиката *подобається* змінюється (розділ *predicates*) після зіставлення і зв'язування змінних у другій підцілі.

У процесі проведення аналізу параметрів виконується перевірка на предмет того, що всім вихідним змінним, вказаним у голові правила, присвоюються значення у тілі правила. Коли змінна не отримує значення у тілі правила, вона розглядається як та, на яку виконується посилання (*reference*). У такому випадку система перевіряє тип даної змінної, якщо він оголошений як *reference* то проблем не буде, інакше Visual Prolog спробує самостійно перевизначити тип змінної як посилальний (*reference*) і компілятор виводить попереджувальне повідомлення. У разі неспроможності перевизначення типу змінної як посилальний (*reference*), генерується повідомлення про помилку.

У наступному прикладі у цільовому твердженні

```
goal
p(T),T=100,write(T).
```

предикат *p* визивається з вихідним параметром, але у правилі для *p* аргумент *X* є вільним (не зв'язаним)

```
predicates
p(integer)

clauses
p(X):-!.
```

Починаючи з версії 5.2 компілятор по замовчуванню генерує помилку при спробі використання стандартного домена як посилального (*reference*), як у

наведеному вище прикладі. Спроба визвати TestGoal виводить повідомлення помилки типа «*Basic domains becomes reference domains*».

При знаходженні вільної змінної, компілятор виводить відповідне повідомлення. Якщо його проігнорувати, то домен (і всі його підддомени) змінної автоматично буде розглядатися домен з посиланням. У таких випадках рекомендовано оголошувати домен, який буде посилатися на бажаний базовий домен, як показано у прикладі (цілі числа будуть містити не цілі числа, а посилання на цілі числа):

```
domains
refinteger = refinteger integer

predicates
p(refinteger)

clauses
p(_).
```

З використанням посилальних типів подовжується час виконання програми, через додаткові перевірки і уніфікації. Але все таки їх використання полегшує рішення цілого ряду задач. При використанні посилальних типів у Visual Prolog застосовується масив *trail* для запам'ятовування, коли посилальні змінні стають зв'язаними. У масиві кожен запис займає 4 байти (запис не буде поміщатися у масив, якщо між створенням і зв'язуванням змінної немає точок відкату), при необхідності масив автоматично збільшує свій розмір.

2.8. Режими детермінізму

Більшість мов програмування є детермінованими. Це означає, що будь-яка множина вхідних значень викликає єдино можливий набір команд, які використовуються для визначення вихідних значень.

Пролог підтримує *недетермінований висновок*, заснований на *недетермінованих предикатах*. Метою управління *детермінізмом* є скорочення як пам'яті, що використовується, так і часу виконання. Пролог володіє системою детермінізму, що чітко типізується, і системою його перевірки, яка примушує програміста розглядати два аспекти поведінки предикатів/фактів:

- яку кількість рішень предикат може вивести;

- чи може виклик предиката завершитися неуспішно.

У термінах виконання програми Пролог режим детермінізму визначає наступні властивості поведінки предикатів (табл. 2.8):

Таблиця 2.8. Режими детермінізму предикатів

	Кількість рішень, яку може вивести предикат		
	0	1	**>1**
Не може бути успішним	Erroneous {}	Procedure {S}	Multi {S, BP}
Може бути неуспішним	Failure {F}	Determ {F, S}	Nondeterm {F, S, BP}

- чи може предикат бути неуспішний? (*Fail - F*);
- чи може предикат бути успішний? (*Succeed - S*);
- чи встановить Пролог точку повернення (відкату) на виклик цього предиката(*BacktrackPoint – BP*).

Використовуючи ключові слова наведеної таблиці у оголошеннях предикативних доменів, програміст може оголосити для предикатів шість різних режимів детермінізму:

- *Multi* (визначає недетерміновані предикати, які можуть здійснювати повернення назад і генерувати множинні рішення. Не можуть бути неуспішні, як мінімум має одне рішення).
- *Nondeterm* (визначає недетерміновані предикати які можуть здійснювати повернення назад і генерувати множинні рішення. Можуть бути неуспішні).
- *Procedure* (визначає предикати, які названі процедурами. Завжди успішні і не породжують точок відкату. Завжди мають тільки одне рішення).
- *Determ* (визначає детерміновані предикати, які можуть бути успішними, але не можуть мати точок відкату. Має не більше одного рішення).

- *Erroneous* (не породжує рішення і не може закінчитися відмовою *fail*. Використовують для управління помилками. Пролог надає вбудовані предикати *exit* і *errorexit*).
- *Failure* (не породжує рішення, але може завершитися відмовою *fail*, тобто неуспішно. Використовується для примушення пошуку з поверненням до найближчої точки відкату).

Запитання. Завдання

1. Як пояснити термін «формальна система» мови програмування Prolog?
2. Що називають «літералом» у логіці висловів?
3. Яку роль відіграють диз'юнкти Хорна у логічному програмуванні?
4. Які основні логічні операції виконують з логічними змінними?
5. Що являється конструкціями синтаксису мови логіки як мови формальної теорії?
6. Який вираз називають «термом»?
7. Яка відмінність між поняттями «предикат» і « відношення»?
8. Що називають «областю дії квантора»?
9. Що собою представляє «метод резолюцій» в обчисленні предикатів?
10. Що таке «правила» і «факти» у Пролозі?
11. У чому заключається ідея логічного програмування?
12. Що таке «арність предиката»?
13. З яких типів фраз складаються пролог-програми?
14. Чи можна зберегти інформацію, привласнивши значення змінній у пролог-програмі?
15. Що означає «зіставлення « у Пролог і чого конкретно стосується?
16. Які основні розділи пролог-програм?
17. Які вимоги до розміщення у програмах предикатів з одним ім'ям?
18. Основні директиви компілятора.
19. Які шаблони потоку параметрів застосовують у пролог-програмах?

20. Що являється формулою формальної системи мови логічного програмування Visual Prolog?

Тест для самоконтролю знань з розділу 2[F]

1. «Позитивний літерал» у логіці висловів логічної мови програмування означає ...
   а) ціль;
   б) логічне заперечення зміни;
   в) змінну.
2. З чого складається алфавіт мови обчислення виразів у логіці висловів логічної мови програмування?
   а) із об'єктів визначених і стандартних типів даних;
   б) із змінних, логічних зв’язків і дужок;
   в) з алфавітних символів;
   г) з алфавітних символів і цифр.
3. Що означає логічна операція «Імплікація», яка виконується над логічними змінними?
   а) протилежність;
   б) логічне «і»;
   в) логічне «або»;
   г) наближена до виразу «то ...якщо».
4. «Логічними формулами» називають ...
   а) вирази, які отримують при використанні логічних змінних та логічних зв’язків;
   б) вирази, які отримують при використанні логічних змінних та об’єктів;
   в) вирази, які отримують із об’єктів визначених і стандартних типів даних;
   г) фрази з наділенням значення істинності змінним.
5. «Аксіоми системи» мови логічного програмування Visual Prolog?
   а) скінченна множина відносин між формулами системи;
   б) складають правила і цілі;
   в) визначена множина формул системи;
   г) зіставлення вільних змінних або їх логічне заперечення.
6. Синтаксис мови логічного програмування Visual Prolog включає?
   а) предметні змінні і константи; терми; вирази;
   б) функціональні і предикатні символи;
   в) логічні зв’язки; квантори; формули;
   г) всі відповіді вірні.
7. Як створюється атомна (елементарна) формула в теорії логіки предикатів?
   а) шляхом застосування предиката до термів P(t1;...;tn), де P - n-вмісний предикативний символ; t1,...,tn - терми;

[F] Усі питання мають один правильний варіант відповіді.

б) шляхом застосування предиката до термів P(t1,...,tn), де P - n-вмісний предикативний символ; t1,...,tn – терми.

8. Як представляються об’єкти у Пролог-програмах?
   а) у вигляді термів, тобто всяка змінна або константа є терм;
   б) у вигляді атомів, тобто всяка змінна або константа є атом;
   в) у вигляді літералів, тобто всяка змінна або константа є літерал;
   г) у вигляді формул, тобто всяка змінна або константа є формула.
9. Диз’юнкція, записана у вигляді імплікації в Пролог-програмах називається?
   а) ціллю;
   б) фактом;
   в) фразою (клаузой).
10. Предикат представляє собою ...
   а) константу;
   б) змінну;
   в) складений об’єкт;
   г) індикатор відношення.
11. Змінні у Прологі використовують для ...
   а) задання загальних фактів, правил, запитів;
   б) оголошення доменів;
   в) тільки для складання запитів(цілі);
   г) використовують тільки в правилах.
12. Як привласнюється значення змінним у Прологі?
   а) об’являється у фактах;
   б) в розділі clauses з використанням оператора привласнення;
   в) шляхом ініціалізації при зіставленні з константами у фактах і правилах;
   г) об’являється в розділі constans.
13. Змінні у Пролог-програмах ...
   а) використовуються як сховище інформації;
   б) використовуються як частина процесу пошуку решення;
   в) використовуються тільки для пошуку з поверненням;
   г) тільки для складання цілі.
14. Розділ Clauses - це розділ ...
   а) в якому оголошуються предикати і домени їхніх аргументів;
   б) в якому записуються факти і правила;
   в) в яком оголошуються типи всіх аргументів, що використовуються;
   г) в якому розміщують прості або складені цілі.
15. Процедурою називається ...
   а) послідовність фраз, які описують об’явлені предикати;
   б) послідовність фраз, які описують один предикат;
   в) процес виконання складених підцілей;
   г) процес узгодження тіла правила з його заголовком.
16. Уніфікація – це …

а) процес збереження результатів пошуку;
б) процес ініціалізації повтору з поверненням під час пошуку всіх рішень;
в) процес пошуку, який виконується під час зіставлення визову з підцілі, і фрази в розділі clauses;
г) процес виключення з розгляду альтернативних підцілей правила в розділі clauses.

17. Цільове твердження вважається доказаним, якщо ...?
а) знайдені відповідні факти;
б) відповідні факти найдені для кожної підцілі листової вершини дерева цілей;
в) знайдені відповідні факти і змінній присвоєно значення;
г) тільки коли ціль узгоджена з тілом правила.

18. Призначення системи детермінізму Прологу?
а) здатність системи за будь-яких умов виводити множинні рішення;
б) здатність системи за будь-яких умов отримувати рішення;
в) визначає властивості поведінки предикатів.

19. Яка мета управління детермінізмом у Пролозі?
а) скорочення пам’яті, що використовується, і часу виконання;
б) здатність зберізати проміжні результати;
в) можливість змішувати домени об’єктів.

20. У якому випадку Пролог підтримує автоматичне перетворення типів?
а) між рядками (string) і ідентифікаторами (symbol);
б) між цілими(integer), дійсними(single);
в) між цілими(integer), дійсними(single) і символами (char);
г) всі відповіді вірні.

РОЗДІЛ 3. УПРАВЛІННЯ ПОРЯДКОМ ВИКОНАННЯ РЕЗОЛЮЦІЙ

У розділі викладено основний вбудований механізм виконання Пролог-програм – пошук з поверненням та методи, які можна використовувати для управління пошуком рішень цільових тверджень, внаслідок чого підвищується ефективність виконання програм.

3.1. Пошук з поверненням

Пролог-програми – це обмежений набір заданих фактів і правил. Однією з найважливіших особливостей Прологу є те, що на додаток до логічного пошуку відповідей на поставлені питання, він може мати справу з альтернативами і знаходити всі можливі рішення. Замість звичайної роботи програми від початку до її кінця, Пролог може повертатися назад і переглядати більше одного шляху при рішенні всіх складових частин завдання.

Механізм пошуку з поверненням у Visual Prolog працює за наступними правилами:

- підцілі повинні бути узгоджені по порядку, зверху вниз;
- предикатні фрази перевіряються в тому порядку, в якому вони з'являються в програмі, зверху вниз;
- коли підціль відповідає заголовку правила, далі повинно бути узгоджено тіло цього правила: тіло правила тепер утворює нову множину підцілей для узгодження;
- цільове твердження вважається узгодженим, коли відповідний факт знайдений для кожного краю (листа) цільового дерева.

Досліджуємо наступну програму

```
domains
        name = symbol

predicates
  подобається(name,symbol)-nondeterm(o,i),(i,i)
  має_авто(name)-nondeterm(o),(i)
  допитливий(name)-nondeterm(i)

clauses
  подобається(ігор,подорожувати).
```

```
подобається(петро,літати).
подобається(степан,читати).
подобається(іван,подорожувати).
подобається(сергій,телебачення).
подобається(ігор,читати).
має_авто(ігор).
допитливий(ігор).
подобається(Хто,подорожувати):-
                                    має_авто(Хто),
                                    допитливий(Хто).
```

Для цільового твердження, що складається з двох підцілей,

```
goal
подобається(X,подорожувати),подобається(X,читати).
```

визначений наступний алгоритм узгодження:

- в першу чергу Visual Prolog відзначає, які підцілі узгоджувалися, а які ні;
- у прикладі Visual Prolog знаходить фразу, відповідну першому факту, що визначає предикат *подобається*;
- підціль *подобається(X,подорожувати)* відповідає факту *подобається(ігор,подорожувати)*, і *X* зв'язується із значенням *ігор*;
- Visual Prolog намагається погоджувати наступну справа підціль. Звернення до другої підцілі починає абсолютно новий пошук з умовою: *X= ігор*;
- перший факт *подобається(степан,читати)* не відповідає підцілі *подобається(X,читати)*, тому що «*читати*» ≠ «*літати*»;
- повинен перевірити наступну фразу, але змінна *X* зв'язана з *ігор* і не відповідає значенню *степан*, так що пошук продовжується;
- у наступних двох фразах немає збігів;
- остання фраза *подобається(ігор,читати)* процедури задовольняє другу підціль цільового твердження;
- слід узгодити підціль *подобається(X,подорожувати)* з виклику із заголовком правила;
- змінна *X* з виклику уніфікується зі змінною *Хто* у правилі і прирівнює її значення «*ігор*»;

- коли підціль відповідає заголовку правила, далі повинно бути узгоджено тіло цього правила (кожна підціль *має_авто(Хто)*, *допитливий(Хто)*), де змінна *Хто* пов'язана з *ігор*. Visual Prolog шукатиме такі факти (*має_авто(ігор)* та *допитливий(ігор)*);
- початкове цільове твердження узгоджене.

Visual Prolog використовує результат процедури пошуку по-різному, залежно від того, як був початий пошук:

- якщо цільове твердження є зверненням з підцілі в тілі правила, то Visual Prolog намагається погоджувати наступну підціль в правилі, після того, як звернення повернене;
- якщо цільове твердження є запитом користувача, то Visual Prolog безпосередньо відповідає: *X=ігор 1 Solution*.

Як видно з програми (див. лістинг), одного разу погодивши цільове твердження, Test Goal повертається (відкочується) назад для пошуку всіх альтернативних рішень. Test Goal повертається назад і в тому випадку, якщо підціль не виконується, сподіваючись переузгодити попередню підціль з іншими фразами.

Виконуючи підціль, Visual Prolog починає пошук з першої фрази, яка визначає предикат. Алгоритм пошуку може різнитися.

Visual Prolog знаходить відповідну фразу:

- якщо є інша фраза, яка, можливо, може знов погоджувати підціль, Visual Prolog виставляє покажчик (з тим, щоб відзначити точку повернення) і зв'язує всі вільні змінні в підцілі (які відповідають значенням у фразі) з відповідними значеннями;
- якщо дана фраза є заголовком правила, то потім оцінюється тіло цього правила. Підцілі в тілі правила повинні бути задоволені для успішного завершення звернення.

Якщо Visual Prolog не може знайти відповідну фразу, тоді цільове твердження не узгоджується і Visual Prolog виконує пошук з поверненням у спробі знов погоджувати попередню підціль. Коли процес досягає останньої

точки повернення, Visual Prolog звільняє всі змінні, яким були привласнені нові значення (після того, як була поставлена точка повернення), і знов намагається погодити початкове звернення.

Коли Visual Prolog виконує повернення до звернення, новий процес пошуку починається з точки відкату, що була виставлена останньою. Якщо пошук безуспішний, то знов виконується пошук з поверненням. Якщо процес пошуку з поверненням вичерпав всі фрази для всіх підцілей, то це означає, що цільове твердження не узгоджується.

3.2. Механізм управління пошуком у Visual Prolog

Вбудований механізм пошуку з поверненням у Visual Prolog може привести до пошуку непотрібних рішень, витрачається більше часу на виведення результатів, внаслідок чого втрачається ефективність у випадках, коли бажано знайти тільки одне рішення. І навпаки, може виявитися необхідним продовжувати пошук додаткових рішень, навіть якщо цільове твердження вже узгоджене.

У Пролозі існують деякі методи, які можна використовувати для управління пошуком рішень цільових тверджень, він забезпечує два інструментальні засоби, які дають можливість управляти механізмом пошуку з поверненням. Це предикат *fail*, який використовується для ініціалізації пошуку з поверненням, і предикат *cut* або *відсікання* (позначається «!») – для заборони можливості повернення.

Visual Prolog починає пошук з поверненням у разі невдалого завершення виклику. У певних ситуаціях буває необхідно ініціалізувати виконання пошуку з поверненням, щоб знайти альтернативні рішення. Visual Prolog підтримує спеціальний предикат *fail*, що викликає неуспішне завершення, і, отже, ініціалізує повернення. Якщо до наведеного коду Пролог-програми поставити просту ціль (як було розглянуто у попередніх прикладах)

```
domains
  name = symbol

predicates
  столиця(name, name)
```

```
clauses
 столиця("Київ","України").
 столиця("Варшава","Польщі").
 столиця("Мінськ","Білорусії").
 столиця("Кишинів","Молдови").
 % необхідно знайти всі рішення цілі

goal
 столиця (Столиця,_країни).
```

то Test Goal знайде всі рішення цілі столиця (Столиця,_країни) і відобразить значення всіх змінних у вигляді:

```
Столиця=Київ, _країни=України
Столиця=Варшава, _країни=Польщі
Столиця=Мінськ, _країни=Білорусії
Столиця=Кишинів, _країни=Молдови
4 Solutions
```

У випадках використання цільового твердження, наприклад, *результат* або *run* (без аргументів), слід створити правило для його виконання та виведення результатів пошуку (використання предиката *write*).

```
domains
 name = symbol

predicates
 столиця(name, name)
 результат()

clauses
 столиця("Київ","України").
 столиця("Варшава","Польщі").
 столиця("Мінськ","Білорусії").
 столиця("Кишинів","Молдови").
 результат:-столиця(X,Y),
 write(X," столиця",Y,"\n").
 %необхідно знайти всі рішення цілі

goal
 результат.
```

Після того, як цільове твердження *результат* з розділу goal, виконане вперше, ніщо не говорить Прологу про необхідність продовження пошуку з поверненням. Тому звернення до предиката *столиця* приведе тільки до одного рішення.

```
Київ столиця України
yes
```

Як же знайти всі можливі рішення? Наступний приклад ілюструє використання спеціального предиката *fail*, який завжди неуспішний та ініціює пошук з поверненням для знаходження всіх існуючих рішень. Його слід розміщувати останньою підціллю у тілі правила

```
результат:-столиця(X,Y),
write(X," столиця ",Y,"\n"), fail.
```

Результатом пошуку є список всіх можливих рішень для предиката *столиця*:

```
Київ столиця України
Варшава столиця Польщі
Мінськ столиця Білорусії
Кишинів столиця Молдови
no
```

Так як *fail* завжди неуспішний, список завершується *no*. Щоб отримати успішне завершення Пролог-програми слід у розділі *clauses* розмістити факт *результат()*, який підтверджуватиме цільовий запит.

```
clauses
 столиця("Київ","України").
 столиця("Варшава","Польщі").
 столиця("Мінськ","Білорусії").
 столиця("Кишинів","Молдови").
 результат:-столиця(X,Y),
                write(X,"столиця",Y,"\n"),fail.

 результат.
goal
 результат.
```

У такому разі програма завершиться успішно *yes*.

Предикат *результат* у програмі використовує *fail* для підтримки пошуку з поверненням, примушуючи Пролог виконувати пошук з поверненням крізь тіло правила

```
результат:-столиця(X,Y),
write(X," столиця ",Y,"\n"),fail.
результат.
```

Fail не може бути узгоджений, він завжди неуспішний, тому Visual Prolog вимушений повторювати пошук з поверненням. При пошуку з поверненням він повертається до останнього звернення, яке може вивести множинні рішення. Таке звернення називають *недетермінованим*. Недетерміноване звернення є

протилежністю *детермінованому* зверненню, яке може вивести тільки одне рішення.

Предикат *write* у даному прикладі не може бути знов узгоджений (він не може запропонувати нових рішень), тому Visual Prolog повинен виконати відкат далі, цього разу до першої підцілі у правилі. Предикат *fail* весь час завершується невдало – немає можливості для досягнення підцілі, розташованої після *fail*.

Ціль *run* у наступному прикладі виводить результат по першому правилу, після натиснення будь-якої клавіші – виводить *Yes* (підтвердження успішного завершення програми).

```
nowarnings

predicates
  run - nondeterm ()

clauses
  run:-
  write("******Пролог-програма*******"),nl,
  write("Перший рядок повідомлення"),nl,
  readchar(_).
  run:-
  write("Другий рядок повідомлення "),nl,
  readchar(_).

goal
  run.
```

Результат виконання програми за першим правилом:

```
******Пролог-програма*******
Перший рядок повідомлення
yes
```

Якщо останньою підціллю першого правила розмістити предикат *Fail*, то буде виконане і наступне правило:

```
clauses
  run:-
  write("******Пролог-програма*******"),nl,
  write("Перший рядок повідомлення"),nl,
  readchar(_),fail.
  run:-
  write("Другий рядок повідомлення "),nl,
  readchar(_).
```

Пошук у такій пролог-програмі виконується за всіма правилами:

```
******Пролог-програма*******
Перший рядок повідомлення
Другий рядок повідомлення
Yes
```

nowarnings – спеціальна директива транслятора (*ignore*), яка подавляє дані у випадку, коли змінна використовується у фразі тільки один раз. Директива *nowarnings* включається у код тільки для завершування дії у випадках:

- якщо змінна використовується тільки раз;
- або це помилка;
- або звичайну змінну треба замінити анонімною змінною;
- або змінна починається с символу підкреслення, тобто виконується абстрактний зв'язок доменів.

Readchar використовується в програмах тексту, коли вхідним пристроєм є клавіатура. Перший викликаний *readchar* не одержує перший надрукований символ доти, доки клавіша *[Enter]* не буде натиснута. Тільки тоді вхідний потік стає доступним *readchar*. Від цього моменту всі знаки, надруковані перед введенням, стають негайно доступними викликаним *readchar* предикатам (якщо такі є). Коли всі надруковані (перед введенням) символи використовували вхідні потоки, тоді викликаний *readchar* буде очікуванням введення.

Наступний приклад демонструє роботу предиката *fail* при обчисленні формул, які описані у чотирьох правилах,

```
domains
 product = real

predicates
 multiply1(product,product,product) - procedure (i,i,o)
 multiply2(product,product,product) - procedure (i,i,o)
 multiply3(product,product,product,product) - procedure (i,i,i,o)
 multiply4(product,product,product,product) - procedure (i,i,i,o)
 pi(real,symbol) - nondeterm (o,i)
 radius(real,symbol) - nondeterm (o,i)
 high(integer,symbol) - nondeterm (o,o),nondeterm (o,i)
 run()- multi()

clauses
 multiply1(X,Y,Product1):-
 Product1=4*X*(Y*Y).
 multiply2(Q,W,Product2):-
 Product2=(4/3)*Q*(W*W).
```

```
multiply3(E,R,T,Product3):-
Product3=(1/3)*E*(R*R)*(3*T-R).
multiply4(I,O,P,Product4):-
Product4=2*I*O*P.

pi(3.14,pi).
radius(7.3,"R").
high(11,h).

run():-pi(X,pi),radius(Y,"R"),multiply1(X,Y,Product1),
        write("Площа поверхні сфери ","= ",Product1),nl,fail.
run():-pi(Q,pi),radius(W,"R"),multiply2(Q,W,Product2),
        write("Об'єм кулі ","= ",Product2),nl,fail.
run():-pi(E,pi),high(R,h),radius(T,"R"),multiply3(E,R,T,Product3),
        write("Об'єм сегмента кулі ","= ",Product3),nl,fail.
run():-pi(I,pi),high(O,h),radius(P,"R"),multiply4(I,O,P,Product4),
        write("Бокова поверхня кулі ","= ",Product4),nl.
goal
run.
```

і виведенні результатів обчислення:

```
Площа поверхні сфери = 669.3224
Об'єм кулі = 223.1074667
Об'єм сегмента кулі = 1380.448667
Бокова поверхня кулі = 504.284
Yes
```

Ще один приклад отримання множинних рішень з використанням умови.

```
domains
студент = string
номер_списку = word

predicates
список(номер_списку, студент) - nondeterm(o,o)
run()

clauses
список(1,"Береговий В.").
список(2,"Бєловодський В.").
список(3,"Братко Н.").
список(4,"Глеб А.").
список(5,"Деркач А.").
список(6,"Зимогляд С.").
список(7,"Ігнетенко М.").

список(8,"Ільченко Д.").
список(9,"Коральков М.").
список(10,"Лутченко А.").
список(11,"Марчук А.").
список(12,"Мельниченко А.").
список(13,"Палій В.").
список(14,"Пилипенко О.").
```

```
список(15,"Потапов Ю.").
список(16,"Приходько Я.").

список(17,"Стеба Н.").
список(18,"Фуштей Б.").
список(19,"Черняєв А.").
список(20,"Черноморець О.").
список(21,"Шуліка Д.").

run:-write("***Список студентів з 1 по 7***"),
nl,fail.

run:-список(Номер,Студент),
Номер <= 7,
Номер >= 1,
write("\t",Студент),
nl,fail.

run:-write("***Список студентів з 8 по 16***"),
nl,fail.

run:-список(Номер,Студент),
Номер <= 16,
Номер >= 8,
write("\t",Студент),
nl,fail.

run:-write("***Список студентів з 17 по 21***"),
nl,fail.

run:-список(Номер,Студент),
Номер <= 21,
Номер >= 17,
write("\t",Студент),
nl,fail.

run.

goal
run.
/*
***Список студентів з 1 по 7***
    Береговий В.
    Бєловодський В.
    Братко Н.
    Глеб А.
    Деркач А.
    Зимогляд С.
    Ігнетенко М.
***Список студентів з 8 по 16***
    Ільченко Д.
    Коральков М.
```

```
    Лутченко А.
    Марчук А.
    Мельниченко А.
    Палій В.
    Пилипенко О.
    Потапов Ю.
    Приходько Я.
***Список студентів з 17 по 21***
    Стеба Н.
    Фуштей Б.
    Черняєв А.
    Черноморець О.
    Шуліка Д.
yes*/
```

3.3. Відсікання

Visual Prolog передбачає можливість ***відсікання***, яка використовується для переривання пошуку з поверненням. Відсікання позначається знаком, оклику (*!*), через нього неможливо ініціалізувати пошук з поверненням.

Відсікання поміщається в програму таким же чином, як і підціль в тілі правила, може розміщатися на будь-якому (як вимагає логіка) місці. Коли процес проходить через відсікання, негайно задовольняється звернення до *cut* і виконується звернення до наступної підцілі (якщо така існує). Одного разу пройшовши через відсікання, вже неможливо провести відкат до підцілей, розташованих перед відсіканням у фразі, що обробляється. Також неможливо повернутися до інших фраз, що визначають предикат, який обробляється (предикат, що містить відсікання).

Випадки застосування відсікання:

- якщо наперед відомо, що певні посилання ніколи не приведуть до осмислених рішень (пошук рішень у цьому випадку буде зайвою витратою часу), – застосується так зване «*зелене*» відсікання. Програма стане працювати швидше і буде економитись пам’ять.
- якщо сама логіка програми вимагає застосування відсікання для виключення альтернативних підцілей використовують «*червоне*» відсікання.

Наприклад, у наступній програмі Пролог знайде за правилом і виведе список стандартних програм.

```
predicates
 програма(symbol)
 стандартна(symbol)
 ms_office(symbol)
 список(symbol)

clauses
 програма(провідник).
 програма(word).
 програма(блокнот).
 програма(калькулятор).
 програма(excel).
 програма(paint).
 програма(powerpoint).
    стандартна(провідник).
    стандартна(блокнот).
    стандартна(калькулятор).
    стандартна(paint).
 ms_office(word).
 ms_office(excel).
 ms_office(powerpoint).
 список(X):-програма(X),стандартна(X).

goal
 список(Програма).
```

Для цільового твердження *список(Програма)* Пролог знаходить всі можливі рішення

```
Програма=провідник
Програма=блокнот
Програма=калькулятор
Програма=paint
4 Solutions
```

Для виключення альтернатив пошуку слід у правило додати предикат *cut*.

```
список(X):-програма(X),стандартна(X),!.
```

Результатом пошуку у такому разі буде одне рішення: *Програма=провідник*.

Якщо відсікання розмістити другою підціллю у правилі,

```
список(X):-програма(X) ,!,стандартна(X).
```

результат пошуку буде таким же, тому що у процедурі факт *програма(провідник)* відноситься до стандартних програм. Але якщо переставити місцями факти у розділі

```
clauses
 програма(word).
 програма(провідник).
```

тоді Пролог не знайде жодного рішення *No Solution* (перевіривши факт *програма(word)*, який відноситься до офісних програм, Пролог зупиняє пошук).

3.4. Запобігання пошуку з поверненням до попередньої підцілі в правилі

У прикладах, що показують, як слід використовувати відсікання, розглядаються декілька умовних правил (*r1, r2 і r3*), які визначають умовний предикат *r*, а також декілька підцілей – *a, b, c* і т. д.

Такий запис є способом повідомити Visual Prolog про те, що користувача задовольнить перше рішення, знайдене ним для підцілей *a* і *b*. Маючи можливість знайти множинні рішення при зверненні шляхом пошуку з поверненням, Пролог при цьому не може провести пошук з поверненням через відсікання і знайти альтернативне рішення для звернень *a* і *b*. Він також не може повернутися до іншої фрази, що визначає предикат *r1*.

У наступному прикладі відсікання в правилі *придбати_авто* означає, що оскільки в базі даних міститься тільки один автомобіль «*Корвет*» приємного кольору, хоч і з дуже високою ціною, то немає потреби шукати іншу машину. Цільове твердження «знайти *corvette* приємного кольору», з критерієм (*<25000*) по вартості:

```
predicates
  придбати_авто(symbol, symbol)
  авто (symbol,symbol,integer)
  колір(symbol,symbol)

clauses
  придбати_авто(Модель,Кол):-
                                  авто(Модель, Кол,Вартість),
                                  колір (Кол, приємний), !,
                                  Вартість < 25000.
  авто(maserati,зелений,25000).
  авто(corvette,чорний,24000).
  авто(corvette,червоний,26000).
  авто(porsche,червоний,24000).
  колір(червоний,приємний).
  колір(чорний,класичний).
  колір(зелений,яскравий).

goal
  придбати_авто(corvette,Колір).
```

Отримавши цільове твердження *придбати_авто(corvette,Колір)*, програма виконується в наступній послідовності:

- Visual Prolog звертається до предиката *авто*, з першої підцілі в тілі правила для предиката *придбати_авто*;
- виконує перевірку, яка завершується невдало, у факті *авто(maserati,зелений,25000)* для першої машини, *Maserati*;
- потім перевіряє наступну фразу *авто(corvette,чорний,24000)* і знаходить відповідність, зв'язуючи змінну *Колір* із значенням *чорний*;
- переходить до наступного звернення і перевіряє, чи у зазначеної машина приємний колір. Чорний колір у фактах програми не відповідає умові, тому перевірка завершується невдало;
- виконує пошук з поверненням до звернення *авто* і знову шукає *Corvette*, що задовольняє цьому критерію;
- знаходить відповідність *авто(corvette,червоний,26000)* і знову перевіряє колір *колір(червоний,приємний)*. Цього разу колір виявляється приємним, і Visual Prolog переходить до наступної підцілі в правилі: до відсікання. Відсікання негайно виконується, «заморожуючи» всі змінні, раніше зв'язані в цій фразі;
- переходить до наступної (і останньої) підцілі в правилі, до порівняння *Вартість < 25000*.

Перевірка завершується невдало, і Visual Prolog намагається зробити пошук з поверненням з метою знайти іншу модель автомобіля для перевірки. Відсікання запобігає спробі вирішити останню підціль, і цільове твердження завершується невдало *No Solution*

Щоб переконатися, чи виконується остання підціль у правилі, змінимо значення у підцілі *Вартість < 27000*

```
придбати_авто(Модель,Кол):-
                        авто(Модель, Кол,Вартість),
                        колір (Кол, приємний),
                        !,
                        Вартість < 27000.
```

Отримаємо один результат пошуку *Колір=червоний 1 Solution*

Перевіримо, чи виконується остання підціль у правилі, для цього переставимо місцями факти та змінимо умову у правилі на *Вартість > 23000* та поставимо запит «придбати авто будь-якої моделі приємного кольору, вартість якого перевищує 23000 у. о.» *придбати_авто(X,Y).*

```
авто(maserati,зелений,25000).
авто(corvette,чорний,24000).
авто(porsche,червоний,24000).
авто(corvette,червоний,26000).
```

Після знаходження першого рішення Пролог припиняє пошук альтернатив, хоча такі факти є (*авто(corvette,червоний,26000)*):

```
X=porsche, Y=червоний
1 Solution
```

Перевіримо результати пошуку за умови *Вартість > 25000*. У такому випадку перший факт *авто(porsche,червоний,24000)* не відповідає умові правила *No Solution*, альтернативи шукати не треба (відсікання).

Якщо переставити місцями предикати у розділі фраз

```
авто(maserati,зелений,25000).
авто(corvette,чорний,24000).
авто(corvette,червоний,26000).
авто(porsche,червоний,24000).
```

за тієї ж умови отримаємо результат пошуку *X=corvette, Y=червоний* через те, що факт *авто(corvette,червоний,26000)* розміщений першим у списку предикатів, що задовольняють умову правила.

Додамо новий факт *авто(porsche,чорний,23000)* та складемо ще одне правило «взяти у прокат авто (модель) чорного кольору, вартість не важлива, альтернативи не шукати»

```
clauses
 прокат_авто(Модель):-авто(Модель, чорний, _),!.

 придбати_авто(Модель,Кол):-
                                  авто(Модель, Кол, Вартість),
                                  колір (Кол,приємний),
                                  !,
                                  Вартість > 25000.
 авто(maserati,зелений,25000).
 авто(corvette,чорний,24000).
 авто(corvette,червоний,26000).
```

```
авто(porsche,червоний,24000).
авто(porsche,чорний,23000).
колір(червоний,приємний).
колір(чорний,класичний).
колір(зелений,яскравий).
```

для складеної (диз'юнкція) цілі

```
goal
прокат_авто(Z);придбати_авто(X,Y).
```

Пролог знайде рішення

```
Z=corvette
X=corvette, Y=червоний
2 Solutions
```

Щодо першого правила – факти не підтверджуються. Друге правило підтверджується фактами, як вказано у наступній програмі.

```
predicates
приятель(symbol,symbol) - nondeterm (i,o)
дівчина(symbol) - nondeterm (i)
подобається(symbol,symbol) - nondeterm (i,i)

clauses
приятель(іван,ірина):-
        дівчина(ірина),
        подобається(іван,ірина),
        !.
приятель(іван,ігор):-
        подобається(ігор,машина),
        !.
приятель(іван,юлія):-
        дівчина(юлія).
дівчина(тіна).
дівчина(ірина).
дівчина(юлія).
подобається(ігор,машина).
подобається(іван,юлія).

goal
приятель(іван,Хто).
```

Пролог знаходить рішення, а відсікання обмежує їх кількість до одного

```
Хто=ігор
1 Solution
```

Якщо у другому правилі замість відсікання *cut* помістити предикат *fail*, тоді друге правило закінчиться неуспішно

```
приятель(іван,ігор):-
        подобається(ігор,машина),fail.
```

і з'явиться можливість отримати результат за третім правилом

```
Хто=юлія
1 Solution
```

Наступний приклад демонструє пошук пролог і отримання всіх можливих рішень для цілі *run* за правилами:

- Хто є другом Івана, якщо цей Хтось – дівчина та якщо Іван любить цю дівчину;
- Хто є другом Івана, якщо цей Хтось любить машини;
- Хто є другом Івана, якщо ця Хтось – дівчина.

```
predicates
 приятель(symbol,symbol) - nondeterm (i,o)
 дівчина(symbol) - nondeterm (i)
 подобається(symbol,symbol) - nondeterm (i,i)
 run()

clauses
 run:-приятель(іван,X),
        дівчина(X),
        подобається(іван,X),
        write("За 1-м правилом: ",X),nl,fail.
 run:-приятель(іван,Y),
        подобається(Y,машина),
        write("За 2-м правилом: ",Y),nl,fail.
 run:-приятель(іван,Z),
        дівчина(Z),
        write("За 3-м правилом: ",Z),nl.
 дівчина(тіна).
 дівчина(ірина).
 дівчина(юлія).
 подобається(ігор,машина).
 подобається(іван,юлія).
 приятель(іван,ірина).
 приятель(іван,ігор).
 приятель(іван,юлія).

goal
 run.
```

Результат пошуку

```
За 1-м правилом: юлія
За 2-м правилом: ігор
За 3-м правилом: ірина
Yes
```

Приклад. Формула залежності ваги людини від її зросту та віку

```
predicates
людина(symbol,integer,symbol,real)- nondeterm (o,o,i,o)
коеф_ж_до20(real)- nondeterm (o)
```

```
коеф_ж_за20до40(real)- nondeterm (o)
коеф_ж_за40(real)- nondeterm (o)
коеф_м_до20(real)- nondeterm (o)
коеф_м_за20до40(real)- nondeterm (o)
коеф_м_за40(real)- nondeterm (o)
run

clauses
людина(id1,17,ж,165).
людина(id2,35,ж,170).
людина(id3,18,ч,182).
людина(id4,50,ж,155).
людина(id5,30,ч,165).
людина(id6,19,ч,188).
людина(id7,20,ж,178).
людина(id8,55,ч,190).
людина(id9,37,ч,160).
  коеф_ж_до20(0.7).
  коеф_ж_за20до40(0.8).
  коеф_ж_за40(0.9).
  коеф_м_до20(0.72).
  коеф_м_за20до40(0.84).
  коеф_м_за40(0.96).

run:-write("******************Жінки*******************"),nl,fail.
run:-nl,write("жінка до 20 років"),nl,
   людина(Імя,Вік,ж,Зріст),Вік<=20,коеф_ж_до20(K),
   P=(Зріст-100)*K,
   write("\t",Імя," - рекомендована вага: ",P),
   nl,fail.

run:-nl,write("Жінка від 20 до 40 років"),nl,
   людина(Імя,Вік,ж,Зріст),20<Вік and Вік<40,коеф_ж_за20до40(K),
   P=(Зріст-100)*K,
   write("\t",Імя," - рекомендована вага: ",P),
   nl,fail.

run:-nl,write("жінка за 40 років"),nl,
   людина(Імя,Вік,ж,Зріст),Вік>=40,коеф_ж_за40(K),
   P=(Зріст-100)*K,
   write("\t",Імя," - рекомендована вага: ",P),
   nl,fail.

run:-nl,write("******************Чоловіки*******************"),nl,fail.
run:-nl,write("чоловік до 20 років"),nl,
   людина(Імя,Вік,ч,Зріст),Вік<=20,коеф_м_до20(K),
   P=(Зріст-100)*K,
   write("\t",Імя," - рекомендована вага : ",P),
   nl,fail.

run:-nl,write("чоловік від 20 до 40 років"),nl,
   людина(Імя,Вік,ч,Зріст),20<=Вік and Вік<40,коеф_м_за20до40(K),
```

```
    P=(Зріст-100)*K,
    write("\t",Імя," - рекомендована вага : ",P),
    nl,fail.

run:-nl,write("чоловік за 40 років"),nl,
    людина(Імя,Вік,ч,Зріст),Вік>=40,коеф_м_за40(K),
    P=(Зріст-100)*K,
    write("\t",Імя," - рекомендована вага :",P),nl,fail.
run:-!.

goal
run.

/*Рішення:
**********Жінки**********

жінка до 20 років
    id1 - рекомендована вага: 45.5
    id7 - рекомендована вага: 54.6

Жінка від 20 до 40 років
    id2 - рекомендована вага: 56

жінка за 40 років
    id4 - рекомендована вага: 49.5

**********Чоловіки**********

чоловік до 20 років
    id3 - рекомендована вага : 59.04
    id6 - рекомендована вага : 63.36

чоловік від 20 до 40 років
    id5 - рекомендована вага : 54.6
    id9 - рекомендована вага : 50.4

чоловік за 40 років
    id8 - рекомендована вага :86.4
yes
*/
```

3.5. Запобігання пошуку з поверненням до наступної фрази

Відсікання може бути використано, як спосіб повідомити Visual Prolog, що він вибрав вірну фразу для певного предиката.

Наприклад:

```
r(1):- !, a,   b,   c.
r(2):- !, d.
r(3):- !, c.
```

```
r(_):- write("будь-яка фраза.").
```

Використання відсікання робить предикат *r* детермінованим. В даному випадку Visual Prolog виконує звернення до *r* з єдиним цілим аргументом. Припустимо, що проведено звернення *r(1).* Visual Prolog проглядає програму у пошуках відповідності для звернення, знаходить його з першою фразою, що визначає *r*. Оскільки є більш ніж одне можливе рішення для даного звернення, Visual Prolog проставляє точку повернення біля цієї фрази. Тепер Visual Prolog починає обробку тіла правила, проходить через відсікання і виключає можливість повернення до іншої фрази *r*. Це відміняє точки пошуку з поверненням, підвищуючи ефективність виконання програми.

Конструкція такого типа схожа на конструкцію *Select Case* в імперативних мовах програмування, умова перевірки записується у заголовку правил. Можна було б записати такі фрази:

```
r(X):- X=1, !, a,   b,   c.
r(X):- X=2, !, d.
r(X):- X=3, !, c.
 r(_):- write("текст").
```

Проте слід, по можливості, поміщати перевірочну умову саме в заголовок правила, – це підвищує ефективність програми і спрощує її читання.

Складемо програму за даним алгоритмом

```
predicates
  r(integer)-nondeterm(i)
constants
% числові константи, що використовуються у правилах для обчислень
  a=5
  b=10
  c=15
  d=20

clauses
  r(1):-!,F=a+b+c,write(F),nl.
  r(2):-!,Q=d,write(Q),nl.
  r(3):-!,P=c,write(P),nl.
  r(_):-write("Помилка") ,nl.

goal
  /*readint – стандартний предикат для зчитування символів,
  не потребує оголошення. Зчитує символ та передає його у предикат r */
  write("№ 1 до 3: "),
          readint(Номер),
```

```
r(Номер).
```

Приклад, який демонструє вибір дії аналогічно оператору *Select Case:*

```
predicates
   звернення(integer) - nondeterm (i)

clauses
   звернення(1):-
   nl,
   write("Ви обрали 1-е звернення"),nl.
   звернення(2):-
   nl,
   write("Ви обрали 2-е звернення"),nl.
   звернення(3):-
   nl,
   write("Ви обрали 3-е звернення"),nl.
   звернення(X):-
   nl,
   X<>1,X<>2, X<>3,
   write("Такого звернення не існує").

goal
write("Виберіть звернення 1, 2 або 3 (з клавіатури і натисніть [Enter]): "),
readint(X),
звернення(X).
```

Ще один приклад запобігання пошуку з поверненням до наступної фрази (отримання компенсації за повернення квитка на потяг)

```
predicates
 звернення(integer) - nondeterm (i)

constants
 вартість=200    % грн
clauses
 звернення(Час):-nl,Час>12,Компенсація=вартість*0.9,
   write("Компенсація становить ",Компенсація,"грн"),nl.
 звернення(Час):-nl,Час>8, Час<=12,Компенсація=вартість*0.8,
   write("Компенсація становить ",Компенсація,"грн"),nl.
 звернення(Час):-nl,Час>4, Час<=8,Компенсація=вартість*0.7,
   write("Компенсація становить ",Компенсація,"грн"),nl.
 звернення(Час):-nl,Час>1, Час<=4,Компенсація=вартість*0.6,
   write("Компенсація становить ",Компенсація,"грн"),nl.
 звернення(Час):-nl,Час>0,Час<=1, Компенсація=вартість*0.5,
   write("Компенсація становить ",Компенсація,"грн"),nl.

goal
 write("Отримання компенсації за залізничний квиток. \n”),
 write("Введіть термін здачі квитка в годинах "),
 readint(X),
 звернення(X).
```

Результат пошуку, наприклад, за умови повернення квитка за 7 годин до відправлення

```
Отримання компенсації за залізничний квиток.
Введіть термін здачі квитка в годинах:7
Компенсація становить 140грн
Годин_до_відправки=7
1 Solution
```

3.6. Заперечення у Visual Prolog

Для реалізації заперечень у пролог-програмах застосовують предикат *not*, який буде успішним, якщо не може бути доведена істинність даної підцілі. Наприклад для оголошених у програмі фактів

```
facts
 xpositive(symbol,symbol)
 xnegative(symbol,symbol)
```

складені правила для перевірки та однозначного вибору необхідних об'єктів предикатів

```
positive(X,Y):-
  xpositive(X,Y),!.
positive(X,Y):-
  not(xnegative(X,Y)),
  ask(X,Y,yes).

negative(X,Y):-
  xnegative(X,Y),!.
negative(X,Y):-
  not(xpositive(X,Y)),
  ask(X,Y,no).
```

Неможливо виконати скріплення незв'язаних змінних усередині предиката *not*. У випадку виклику підцілі з вільними змінними з середини *not*, Visual Prolog виведе повідомлення про помилку: «*Free variables not allowed in not or retractall*» («Вільні змінні не дозволені в *not* або *retract»*). Це відбувається унаслідок того, що для скріплення вільних змінних в підцілі, підціль повинна уніфікуватися з якою-небудь іншою фразою і виконуватися. Правильним способом управління незв'язаними змінними підцілі у середині є використання анонімних змінних.

Приклад правильного застосування предиката *not* . Пошук студентів, які не мають заборгованості по сесії

```
predicates
  групи(symbol,symbol) - nondeterm (o,o)
  борг(symbol,symbol) - nondeterm (i,i)
  робота(symbol,symbol) - nondeterm (i,i)
  іспит(symbol,symbol) - nondeterm (i,i)
  виконати() - nondeterm()

clauses
  група(ляпунов,група1).
  група(куров,група1).
  група(філяєв,група1).
  група(малюта,група2).
  група(крилов,група3).

  борг (ляпунов,ні).
  борг (куров, ні).
  борг (філяєв, так).
  борг (малюта, так).
  борг (крилов, ні).

  робота (ляпунов, так).
  робота (філяєв, ні).
  робота (малюта, так).
  робота (крилов, так).

  іспит (ляпунов, так).
  іспит (куров, так).
  іспит (філяєв, так).
  іспит (малюта, ні).
  іспит (крилов, так).

  виконати:- група(Студент,_),
             борг (Студент, так),
             write("Студент=",Студент),
             nl,
             fail.
  виконати ().

goal
  write("Список студентів, які не мають боргів по сесії: "),
  nl, виконати.
```

Таких фактів у програмі два

```
Список студентів, які не мають боргів по сесії:
Студент=філяєв
Студент=малюта
yes
```

Для того, щоб Пролог за даним правилом знайшов альтернативні рішення, тобто «Список студентів, які мають заборгованість по сесії»,

```
виконати:- група(Студент,_),
```

```
        not (борг (Студент,так)),
        write("Студент=",Студент),nl,
        fail.
```

Змінна *Студент* зв'язується до того, як Visual Prolog робить висновок, що фраза *борг(Студент,так)* не є істиною. Дана фраза працює коректно. Результат

```
Список студентів, які мають заборгованість по сесії:
Студент=ляпунов
Студент=куров
Студент=крилов
yes
```

Для прикладу ще одна коректна фраза використання заперечень:

```
виконати:- група(Студент, група1),
        not (борг (Студент,так)),
        write("Студент=",Студент),nl,
        fail.
```

Результатом пошуку осіб зі студентів з *група1* є

```
Список студентів, які мають борги по сесії:
Студент=ляпунов
Студент=куров
```

Некоректною буде фраза, коли предикат *not* помістити у першу підціль, коли змінна *Студент* ще буде вільною

```
виконати:- not (група(Студент, група1)),
        борг (Студент,так),
        write("Студент=",Студент),nl,
        fail.
```

тоді Пролог виводить повідомлення

```
«E;Test_Goal, pos: 666, 704 Free variable are not allowed in 'not' or 'retractall'»
```

Навіть коли в *not*(*група(Студент, група1)),* замінити змінну *Студент* на анонімну змінну *not*(*група(_, група1))*, – фраза не повертатиме помилку, але результат буде не вірним

```
Список студентів, які не мають боргів по сесії:
yes
```

Ця фраза стверджує, що не треба шукати будь-якого студента з групи (тільки не з *група1*), які мають заборгованості по сесії. Пролог виводить *yes*, тому що існує факт *виконати ()*.

Невірне використання предиката *not* приведе до повідомлення про помилку або до помилок в логіці програми.

Запитання. Завдання

1. Принцип роботи механізм пошуку з поверненням у Prolog-програмах?
2. Коли цільове твердження вважається узгодженим?
3. У якому порядку розглядаються фрази пролог-програми при узгодженні цільового твердження?
4. Що відбувається із змінними, коли процес пошуку досягає останньої точки повернення?
5. Недоліки механізму пошуку з поверненням.
6. За допомогою яких інструментальних засобів можна управляти механізмом пошуку з поверненням?
7. Предикат Пролог, який завжди неуспішний?
8. Яке звернення називають не детермінованим?
9. Для чого використовується відсікання?
10. Вид детермінізму предиката при використанні відсікання?
11. Випадки застосування предиката заперечення?

Тест для самоконтролю з розділу 3[Ғ]

1. Які методи можна використовувати у Visual Prolog для управління пошуком рішень цільових тверджень?
   а) предикат repeat: - repeat.;
   б) встановлення покажчика в точці повернення;
   в) предикат fail і cut як інструментальні засоби;
   г) всі відповіді вірні.
2. Який предикат використовується для ініціалізації пошуку з поверненням?
   а) предикат !
   б) предикат cut
   в) предикат fail
   г) всі відповіді вірні.
3. Предикат fail …
   а) викликає успішне завершення пошуку рішень;
   б) викликає неуспішне завершення пошуку рішень;
   в) починає пошук з поверненням у разі невдалого завершення виклику;
   г) відсікає альтернативи пошуку рішень.

[Ғ] Усі запитання мають один правильний варіант відповіді.

4. Вимоги до застосування предиката fail?
   а) розміщувати останньою підціллю тільки у правилах;
   б) розміщувати до перевірки умов у правилах;
   в) розміщувати після перевірки умов у правилах;
   г) розміщувати останньою підціллю у фразі.
5. Які дії треба виконати, щоб отримати успішне завершення Пролог-програми, у фразах якої використовується предикат fail?
   а) у розділі predicates оголосити предикат з цільового твердження з режимом determ;
   б) у розділі predicates оголосити предикат з цільового твердження з режимом nondeterm;
   в) у розділі predicates оголосити предикат з цільового твердження з режимом procedure;
   г) у розділі clauses розмістити факт, який підтверджуватиме цільовий запит.
6. Мета застосовування директиви nowarnings у пролог-програмах?
   а) подавляє дані у випадку, коли змінна використовується у фразі тільки один раз;
   б) використовується в програмах тексту, коли вхідним пристроєм є клавіатура;
   в) перевіряє на помилки відповідності типів даних;
   г) всі відповіді вірні.
7. У якому з випадків підвищує ефективність програми і спрощується її читання?
   а) коли перевірочна умова розміщена в заголовку правила;
   б) коли перевірочна умова розміщена у тілі правила;
   в) коли перевірочна умова розміщена у цільовому твердженні;
   г) всі відповіді вірні.
8. У чому особливість застосування предиката not?
   а) усередині предиката не можливо використовувати вільні змінні;
   б) усередині предиката не можливо виконати скріплення незв’язаних змінних;
   в) існує можливість скріплення незв'язаних змінних усередині предиката;
   г) усередині предиката не можливо використовувати змінні.
9. Спосіб управління незв’язаними змінними з середини предиката not?
   а) використання анонімних змінних;
   б) використання вільних змінних;
   в) відсікання;
   г) перевірка умови.
10. Вимоги до розміщення предиката cut?
   а) у будь-якому місці фрази, якщо наперед відомо, що певні посилання ніколи не приведуть до осмислених рішень або цього вимагає логіка програми;

б) у кінці фрази, якщо наперед відомо, що певні посилання ніколи не приведуть до осмислених рішень або цього вимагає логіка програми;

в) у будь-якому місці фрази, якщо наперед відомо, що певні посилання ніколи не приведуть до осмислених рішень;

г) тільки після перевірки умов у будь-якому місці фрази.

РОЗДІЛ 4. АРИФМЕТИЧНІ ОБЧИСЛЕННЯ

Visual Prolog не призначений для програмування задач з великою кількістю арифметичних операцій. Однак мова містить широкий набір вбудованих арифметичних операторів. Пролог дозволяє також порівнювати арифметичні вирази, використовуючи вбудовані предикати.

4.1. Арифметичні вирази

Арифметичний вираз є числом або структурою. У структуру може входити одна або більше компонент, таких, як:

- числа;
- арифметичні оператори;
- арифметичні спискові вирази;
- змінна, конкретизована арифметичним виразом;
- унарні функтори;
- функтори перетворення;
- арифметичні функтори.

Числа. Діапазони чисел, що входять в арифметичні вирази, залежать від реалізації Прологу.

Арифметичні оператори. В будь-яку Пролог-систему можуть включатися всі звичайні арифметичні оператори:

\+ – складання;

\- – віднімання;

* – множення;

/ – ділення;

mod – остача від ділення цілих чисел;

div – цілочисельне ділення.

Арифметичні спискові вирази. Якщо *X* – арифметичний вираз, то список *[X]* також є арифметичним виразом.

Наприклад *[1,2,3]*. Перший елемент списку використовується як операнд у виразі, скажімо,

```
X is ([l,2,3] +5)
```

має значення *6*.

Арифметичні спискові вирази корисні при обробці символів, оскільки останні можуть розглядатися як невеликі цілі числа. Наприклад, символ 'a' еквівалентний *[97]*, що відповідає коду ASCII. Тому значення виразу *'p'+'A'-'a'* дорівнює *80* (для символу *'p'*).

Змінна, конкретизована арифметичним виразом. Наприклад:

X-25+12 та У-25(7+A)*

Унарні функтори. Наприклад:

+(X) та -(У)

Функтори перетворення. Застосовуються в обчисленнях, де використовуються числа з плаваючою крапкою, наприклад:

float (X)

перетворює ціле число *X* в число з плаваючою крапкою.

Арифметичні функтори. Наприклад, квадрат *(X)* оголошений як оператор та еквівалентний арифметичному виразу *(X*X)*.

4.2. Арифметичні оператори

Іноді зручно записувати деякі функтори як оператори. Це особлива форма синтаксису, що полегшує читання відповідних структур.

Наприклад, арифметичні операції зазвичай записуються як оператори. В арифметичному виразі *x + y * z* знаки додавання і множення є операторами. Якщо ж даний арифметичний вираз записати в звичайному для структур вигляді, то він буде виглядати наступним чином: *+(x,*(y, z))*. Однак у деяких випадках операторна форма запису зручніша, тому що структурна форма вимагає поміщення аргументів функтора в круглі дужки, що іноді обтяжливо.

Оператори не викликають виконання будь-яких арифметичних операцій. Так, у Пролозі *3+4* і *7* означають різні об’єкти. Терм *3+4* – інший спосіб запису терма *+(3,4)*, який є структурою.

Читання арифметичних виразів вимагає знання трьох властивостей кожного оператора: його *позиції*, його *пріоритету* та його *асоціативності*.

Синтаксис терма, що містить оператори, частково залежить від їх позицій. Оператори, подібні знакам **+, -, *** і **/**, записуються між своїми аргументами і тому називаються *інфіксними операторами*. Можна також розміщувати оператори перед їх аргументами. Так, у виразі *-x+y* мінус перед *x* позначає арифметичну операцію зміни знака. Оператори, що записані перед своїми аргументами, називаються *префіксними операторами*. Нарешті, деякі оператори можуть розміщуватися після свого аргументу. Наприклад, оператор обчислення факторіала, розміщується після числа, для якого необхідно обчислити факторіал. У математичних позначеннях факторіал *x* записується як *x!* , де знак оклику позначає операцію обчислення факторіала. Оператори, записані після своїх аргументів, називаються *постфіксними операторами*. Таким чином, позиція оператора вказує його місце по відношенню до своїх аргументів.

Для використання операторів необхідні правила, що вказують порядок виконання операцій. Саме цій цілі служить *пріоритет*. Наприклад, структурна форма *+ (x, * (y, z))* явно показує порядок виконання операцій, оскільки структура з функтором * є аргументом структури з функтором +. Для того щоб ЕОМ правильно виконала відповідні обчислення, необхідно спочатку виконати множення – тоді в структурі з + будуть відомі значення аргументів.

Пріоритет оператора визначає, яка операція виконується першою. У Пролозі кожен оператор пов’язаний зі своїм класом пріоритету.

Клас пріоритету являє собою ціле число, величина якого залежить від конкретної версії Прологу.

Однак у будь-якій версії оператор з великим пріоритетом має клас пріоритету, більш близький до 1. Якщо класи пріоритетів приймають значення з діапазону від 1 до 255, то оператор з першим класом пріоритету виконується першим, до виконання операторів, що належать, наприклад, до класу 129. У Пролозі оператори множення і ділення належать до більш високого класу пріоритетів, ніж додавання і віднімання, тому терм *a-b/c* еквівалентний терму – *(a,/(b,c))*. Точна відповідність між операторами і класами пріоритетів в даний

момент не суттєва, проте бажано запам’ятати відносний порядок виконання операцій.

Необхідність знання асоціативності операторів стає очевидною, коли потрібно визначити порядок виконання операторів з однаковим пріоритетом. Наприклад, яким виразам еквівалентний вираз *«8/2/2»* – *«(8/2)/2»* або *«8/(2/2)»*? У першому випадку при інтерпретації виразу було б отримано значення 2, у другому 8. Для того, щоб мати можливість розділити ці два випадки, необхідно знати, чи є даний оператор *лівоасоціативним* або *правоасоціативним*. *Лівоасоціативний* оператор повинен мати зліва операції однакового або нижчого пріоритету, а праворуч – операції нижчого пріоритету. Наприклад, всі арифметичні операції (+, -, * і /) є лівоасоціативними. Це означає, що вирази, подібні «8/4/4», інтерпретуються як *«(8/4)/4»*, а вираз *«5+8/2/2»* еквівалентний *«5+((8/2) / 2)»*.

Таблиця 4.1. Арифметичні оператори

Операнд 1	Оператор	Операнд 2	Результат
Ціле	+, -, *	Ціле	Ціле
Дійсне	+, -, *	Ціле	Дійсне
Ціле	+, -, *	Дійсне	Дійсне
Дійсне	+, -, *	Дійсне	Дійсне
Ціле або дійсне	/	Ціле або дійсне	Дійсне
Ціле	Div	Ціле	Ціле
Ціле	Mod	Ціле	Ціле

На практиці у виразах, розуміння яких ускладнюється правилами пріоритету та асоціативності, треба використовувати круглі дужки.

Атоми +, -, *, /, mod і div – є звичайними атомами Прологу і можуть використовуватися майже в будь-якому контексті. Зазначені атоми – не вбудовані предикати, а *функтори*, що мають силу тільки в межах арифметичних

виразів. Вони визначені як *інфіксні оператори*. Ці атоми є головними функторами в структурі.

Арифметичний оператор виконується таким чином: по-перше, обчислюються арифметичні вирази по обидві сторони оператора; по-друге, над результатом обчислень виконується потрібна операція.

Позиція, пріоритет і асоціативність арифметичних операторів задані і перераховані в таблиці операторів (табл. 4.1).

4.3. Обчислення арифметичних виразів

У Пролозі не допускаються привласнення виду Сума = 2+5. Знак рівності є інфіксним предикатом. Символи в правій частині від знака рівності складають арифметичний вираз.

Арифметичні вирази складаються з *операндів* (чисел і змінних), *операторів* (+, -, *, /, div та mod) і *дужок* (пріоритет операції), наприклад:

```
A = 1 + 6 / (11 + 3) * B
```

Числа у шістнадцятирічній та восьмирічній системі починаються як *"Ox"* або *"Oo"* відповідно, наприклад:

```
OxFFF = 4095
86 = 0o112 + 12
```

Значення виразу може бути обчислено, тільки якщо всі змінні в момент обчислення визначені.

4.4. Порівняння арифметичних виразів

Системні предикати =, < >, >, <, >= і <= визначені як *інфіксні оператори* і застосовуються для порівняння результатів двох арифметичних виразів.

Для предиката @ доказ цільового твердження *X* @ *Y* закінчується успіхом, якщо результати обчислення арифметичних виразів *X* і *Y* знаходяться в такому відношенні один до одного, яке задається предикатом @.

Таке цільове твердження не має побічних ефектів і не може бути погоджено знову. Якщо *X* чи *Y* – не арифметичні вирази, то виникає помилка.

Предикат рівності являється вбудованим, тобто він вже визначений в Пролог-системі і працює так, якби він визначає факт, наприклад *X=X*.

При узгодженні з базою даних, де *X* і *Y* – це будь-які терми, в яких можуть міститися неконкретизовані змінні, діють наступні правила:

- якщо *X* являє собою неконкретизовану змінну, а змінна *Y* конкретизована (значення неважливе), то *X* і *Y* рівні. Крім того, *X* стане конкретизованою – їй буде дано те ж значення, що і *Y*. Наприклад, наступне питання призведе до того, що *X* буде присвоєно значення у вигляді структури: *їхати(клерк, велосипед)*:
  їхати (клерк, велосипед) = X.
- цілі числа і атоми завжди рівні між собою. Наприклад, спроби узгодити такі цільові твердження дадуть результати, показані праворуч:

  папір = папір / * вірно * /
  папір = олівець / * невірно * /
  1066 = 1066 / * вірно * /
  1206 = 1583 / * невірно * /
- Дві структури рівні, якщо вони мають один і той же функтор і однакове число аргументів, причому всі відповідні аргументи рівні. Наприклад, при узгодженні наступного цільового твердження *X* буде присвоєно конкретне значення велосипед:
  їхати (листоноша, велосипед) = їхати (листоноша, X).

Структури можуть бути вкладені одна в іншу на якусь глибину. Якщо такі вкладені структури перевіряються на рівність, перевірка займе більше часу, оскільки необхідно перевірити всі структури. Спроба узгодити наступну ціль:

a (b, C, d (e, F, g (h, i, J))) = a (B, c, d (E, f, g (H, i, j))).

буде успішною, а змінні B, C, F, E, J будуть конкретизовані, їм будуть присвоєні відповідно значення b, c, f, e, j. Що станеться, якщо спробувати прирівняти дві неконкретизовані змінні? Це спеціальний випадок першого з вищенаведених правил. Так, ціль буде узгоджена і дві змінні стануть зв’язаними. Якщо дві змінні зв’язані, то, при конкретизації однієї з них другою змінною, буде автоматично присвоєно те ж саме конкретне значення, що й

першої. Таким чином, в наступному правилі другий аргумент буде конкретизований так само, як перший:

пред (X, Y): - X = Y.

Цільове твердження X = Y завжди вірне (тобто узгоджується з базою даних), якщо один з аргументів неконкретизований. Більш простий спосіб запису такого правила полягає у використанні того факту, що змінна дорівнює самій собі:

пред(X, X).

Пролог пропонує також предикат '< >' відповідний «не дорівнює». Цільове твердження X < > Y вірне в тих випадках, коли не доказове твердження X = Y, і навпаки. Таким чином, X < > Y означає, що X не може бути рівним Y.

За допомогою предикатів описуються наступні відносини (табл. 4.2).

Таблиця 4.2. Інфіксні оператори

X = Y	X дорівнює Y
X < > Y	X не дорівнює Y
X <Y	X менше Y
X> Y	X більше Y
X <= Y	X менше або дорівнює Y
X> = Y	X більше або дорівнює Y

Використання предикатів ілюструють приклади, що наведені у табл. 4.3.

Таблиця 4.3. Використання предикатів

a > 5	закінчується невдачею
1 +2 +7> 3 +2	закінчується успіхом
3 +2 = 5	закінчується успіхом
3 +2 < 5	закінчується невдачею
2 + 1 <> 1	закінчується успіхом
N > 3	закінчується успіхом, якщо змінна N більше 3, і невдачею в іншому випадку.

4.5. Імперативні інструкції

У Visual Prolog 7.4 окрім декларативних інструкцій Прологу додані наступні імперативні інструкції:

- *try catch*;

- *if then else*;
- *foreach do*.

Інструкція *try catch*. Ключові слова *try* та *catch* використовуються разом. За припущенням, коли блок коду може визивати виключення, слід скористатися ключовим словом *try* та використати *catch* для збереження коду, який буде виконаний у разі виникнення виключення. У цьому прикладі в результаті ділення на нуль створюється виключення, яке потім перехоплюється. За відсутності блоків *try* і *catch* виникне збій програми.

Інструкція *if then else*. Використовується для обробки розгалужених процесів. Вона виконує групи інструкцій, вкладених у секції дії «*якщо то або*» за призначенням *then* і *else*. Якщо у групах декілька інструкцій, то вони розділяються комами. Секція *else* може бути відсутня. Виконується за синтаксисом:

```
if (умова)
  then
        Інструкції
  else
        Інструкції
end if
```

Інструкція *foreach*. У перекладі «*для кожного*» призначена для обробки масивів. Вана повторює групу вкладених у неї інструкцій для кожного елементу масиву, використовуючи синтаксис:

```
foreach Ім'я_списка()
   do
        інструкції тіла циклу
end foreach.
```

Якщо тіло містить декілька інструкцій, то вони відділяються одна від одної комою. Як приклад виведення на консоль вмісту масиву цілих чисел:

```
foreach Масив(B)
    do
        M=Масив(B), write(M)
end foreach.
```

4.6. Функції

Visual Prolog включає предикати, які можуть бути використані в якості функції (з повертанням значення), а не звичайні предикати, які повертають значення через вихідні аргументи.

Різниця в них трохи більша, ніж синтаксичні ознаки, оскільки значення, що повертаються, зберігаються в регістрах. Цей механізм дозволяє функціям Prolog повертати значення та отримувати повернене значення.

Оголошення функції виглядає як звичайна декларація предиката, за винятком того, що на початку розміщується доменне ім’я параметра, який повертається:

```
predicates
  unsigned /*Повернути*/ triple(unsigned In)

clauses
  triple(In, Return):- Return = In * 3.

goal
  ReturnTripleVal = triple(6),
  write(ReturnTripleVal).
```

Результат обчислень: *18ReturnTripleVal=18.*

Значення, що повертається, не обов’язково має бути одним із стандартних доменів, це може бути будь-який домен.

Треба пам’ятати, що арифметичні функції мають бути детермінованими (одне рішення), якщо вони використовуються в арифметичних виразах.

Якщо оголошується функція без аргументів, слід ввести пусті дужки для того, щоб відрізнити її від рядка символів, наприклад:

```
predicates
  unsigned година ()

clauses
  година (H): - час (H, _, _, _).
  Година = година (), %  Година =година - запис не вірний
```

Коли предикат оголошений як функція, що повертає значення, він не може бути викликаний як звичайний предикат Prolog за допомогою додаткового аргументу в якості вихідного аргументу, а повинен бути викликаний як

функція. Причиною цього є те, що код, скомпільований до і особливо після виклику функції, відрізняється від коду виклику звичайного предиката.

Приклад функції *neg*, щоб змінювати знак кожного елементу списку:

```
domains
 ilist = integer*

predicates
 Ilist neg(ilist)  % зліва розміщується доменне ім'я параметра функції

clauses
 neg([ ],[ ]).
 neg([Head|Tail],[NHead|NTail]):-
 NHead = -Head,
 NTail = neg(Tail).
```

Приклад використовує не хвостову рекурсію.

Наступний приклад використовує предикат з двома аргументами, що є списками:

```
domains
 ilist = integer*

predicates
 neg(ilist,ilist)   % оголошення предикату

clauses
 neg([ ],[ ]).
 neg([Head|Tail],[NHead|NTail]):-
 NHead = -Head,
 neg(Tail,NTail).
```

і є хвостовою рекурсією. Тому треба бути обережними з застосуванням функцій. Їх основною метою є можливість отримання значень, які повертаються.

Visual Prolog має забезпечувати набір вбудованих математичних функцій і предикатів, які використовують цілі і дійсні значення, це такі як:

val (domain, X) – Явне перетворення між числовими доменами.

X mod Y – Повертає залишок від ділення цілих чисел (модуль) *X* на *Y*.

X div Y – Повертає частку від ділення *X* на *Y*.

abs (X) – Якщо значення *X* – позитивна величина *value*, *abs(X)* повертає це значення, інакше – *1 * value*.

trunc (X) – Усікає *X*, повертає дійсне значення.
exp (X) – Підносить *e* в ступінь *X*.
ln (X) – Логарифм *X* по основі *e*.
log (X) – Логарифм *X* по основі 10.
sqrt (X) – Корінь квадратний з *X*.
sin (X) – Повертає синус свого аргументу.
cos (X) – Повертає косинус свого аргументу.
tan (X) – Повертає тангенс свого аргументу.
arctan (X) – Повертає арктангенс дійсного значення, з яким пов'язаний *X*.
round (X) – Округлює значення *X*, повертає дійсне значення.
random (X) – Привласнює *X* випадкове дійсне число; *0 <= X <1*.
random (X, Y) – Привласнює *Y* випадкове ціле число; *0 <= Y <X*.

Функція *val*(i,i)

Використовує два вхідних параметри (*i,i*). Функция *val* виконує перетворення числових доменів та знаходить будь-які можливі *переповнення*.

Формат: *Result = val(returnDomain, InputValue)*

де *returnDomain* визначає домен, яким конвертує *InputValue*.

Наприклад:

```
res_val_1 = val(integer, 1.7),    %  2
res_trunc_1 = trunc(1.7),         %  усікає до 1
res_val_2 = val(integer, -1.7),   %  -2
res_trunc_2 = trunc(-1.7),        %  усікає до -1
```

Функція cast(i,i)

Функція не виконує ніяких перевірок будь-якого типу, через це будь-який домен може бути дуже необачно перетворений в будь-який інший домен.

Формат: *Result = cast (returnDomain, InputValue)*

Використання *cast* може привести до катастрофічних результатів. При реалізації об'єктів рекомендовано використовувати тільки функцію *val*.

Функція mod(i,i)

Обчислює і повертає залишок від ділення цілих чисел.

Формат: *X mod Y*

Приклади, якщо *Z = X mod Y*:

```
Z = 7 mod 4       %  Z буде дорівнювати 3
Z = -7 mod 4      %  Z буде дорівнювати -3
Z =  4  mod 7     %  Y буде дорівнювати 4
```

Функція div(i,i)

Обчислює і повертає цілу частку від ділення цілих чисел.

Формат: *X div Y*

Приклади, якщо *Z = X div Y*:

```
Z = 7 div 4    % Z = 1
Z = -7 div 4   % Z = -1
Z = 4 div 7    % Z = 0
```

Функція abs(i)

Повертає абсолютне значення свого аргументу.

Формат: *abs(X)*

Якщо *z = abs (X)*, то:

```
Z = abs(-7)   % Z = 7
```

Функція trunc (i)

Усікає число (аргумент типу *real*) праворуч від десяткової крапки, відкидаючи дробову частину.

Формат: *trunc (X)*

Наприклад:

```
Y= trunc(25.45)       % Y = 25
```

Функція exp (i)

Повертає значення *e* в ступені *X*.

Формат: *exp (X)*

Якщо *Y = exp(X)*, то для

```
Y = exp(2.5)   % Z = 12.182493961
```

Функція ln (i)

Повертає значення натурального логарифму від *x* (основа *e*).

Формат: *ln(X)*

Наприклад:

```
Y = ln(12.182493961)      %  Y =2.5
```

Функція log (i)

Повертає значення логарифму по основі *10* від *X*.

Формат: *log (X)*

Наприклад:

```
Y = log(2.5)      %  Y =0.39794000867
```

Функция sqrt (i)

Повертає корінь квадратний від *X*.

Формат: *sqrt (X)*

Наприклад:

```
Y = sqrt (9)      %  Y =3
```

Аргумент у тригонометричних функціях задається у радіанах.

Функція sin (i)

Повертає синус аргументу.

Формат: ***sin*** *(X)*

Наприклад:

```
Pi = 3.141592653,
Y = sin(Pi)   % Y = O
```

Функція cos (i)

Повертає косинус аргументу.

Формат: *cos (X)*

Наприклад:

```
Y = cos(Pi)   % Y = -1
```

Функція tan (i)

Повертає тангенс аргументу.

Формат: *tan (X)*

Наприклад:

```
Y = tan (Pi)   % Y = 0
```

Функція arctan (i)

Повертає арктангенс аргументу.

Формат: arc*tan (X)*

Наприклад:

```
Y =arctan (Pi)   % Y = 1.2 626272556
```

Функція round (i)

Повертає округлене значення аргументу до найближчого цілого.

Формат: *round (X)*

Наприклад:

```
Y = round (5.51)   % Y = 6
Y = round (5.49)   % Y = 5
```

Генератор випадкових чисел

Visual Prolog надає два стандартних предикати для генерації випадкових чисел.

Предикат random (o)

```
random(REAL RandomReal)
```

Ця версія *random/1* повертає випадкове дійсне число, яке задовольняє обмеженням *0 <= RandomReal < 1*.

Наприклад:

```
goal
random(X).        /* X=0.438218271      1 Solution*/

goal
random(X).        /* X=0.36520984623      1 Solution*/

goal
random(X).       /* X=0.16827983629     1 Solution*/
```

Предикат *random*(i,o)

```
random(INTEGER MaxValue, INTEGER RandomInt)
```

повертає випадкове число від *0* до заданого цілого числа *0<=RandomInt<MaxValue*.

Наприклад:

```
goal
random(15,X).         /* X=4    1 Solution*/
```

або

```
goal
random(15,X).        /* X=7  1 Solution*/
```

Крім того, випадкова послідовність нумерації може бути повторно ініціалізована *randominit*

```
randominit(INTEGER Seed)
```

який повторно використовує *random(o)* і *random(i,o)*

```
goal
random(10,X),randominit(10).
```

Приклади

Програма для обчислення виразу $Z = (e^{\sin(x)}) + \sqrt{K + X^2}$:

```
goal
write ("X ="),
readint (X),
nl,
write ("K ="),
readreal (K),
nl,
Z = exp (sin (X)) + sqrt (K + X * X).

  /*Відповідь:
  X=5, K=2, Z=5.579457418
  1 Solution*/
```

У режимі калькулятора обчислити вираз X=(2+5)*3,4, тобто вводячи значення 2, 5, 3.4 з клавіатури:

```
goal
write ("X ="),
readint (X),
nl,
write ("K ="),
readint (K),
nl,
write ("N ="),
readreal (N),
nl,
Z = (X + K) * N.

  /*Відповідь:
  X=2, K=5, N=3.4, Z=23.8
  1 Solution*/
```

Програма, що виконує всі чотири арифметичні операції:

```
predicates
  operation (symbol, real, real)
clauses
  operation ("+", X, Y):-Z = X + Y,
  write (X ,"+", Y ,"=", Z,", виконується ").
  operation ("-", X, Y):-Z = X-Y,
  write (X ,"-", Y ,"=", Z,", виконується ").
  operation ("*", X, Y):-Z = X * Y,
  write (X ,"*", Y ,"=", Z,", виконується ").
  operation ("/", X, Y):-Z = X / Y,
  write (X ,"/", Y ,"=", Z,". виконується ").

goal
  operation (Оператор, 10, 2).

  /*Відповідь:
   10+2=12, виконується Оператор=+
```

```
  10-2=8, виконується Оператор=-
  10*2=20, виконується Оператор=*
  10/2=5. виконується Оператор=/
  4 Solutions*/
```

Варіант реалізації попередньої програми з внутрішнього цільового твердження:

```
predicates
  operation (symbol, real, real)

goal
  write ("Введіть числа [Enter]"),nl,
  readreal (X),nl,
  readreal (Y),
  write ("Результат:"),nl,
  operation ("+", X, Y),
  operation ("-", X, Y),
  operation ("*", X, Y),
  operation ("/", X, Y).

clauses
  operation ("+", X, Y):-Z = X + Y,
  write (X ,"+", Y ,"=", Z), nl.
  operation ("-", X, Y):-Z = X-Y,
  write (X ,"-", Y ,"=", Z), nl.
  operation ("*", X, Y):-Z = X * Y,
  write (X ,"*", Y ,"=", Z), nl.
  operation ("/", X, Y):-Z = X / Y,
  write (X ,"/", Y ,"=", Z), nl.

   /*Введіть числа [Enter]
   10
   5
   Результат:
   10+5=15
   10-5=5
   10*5=50
   10/5=2
   X=10, Y=5
   1 Solution*/
```

Програма обчислення суми ряду: 1 + 2 + 3 + ... + 9 + 10

```
domains
  number,sum=integer

predicates
  summ(number,sum)

  clauses
  summ(11,0):-!.
  summ(Number, Sum):-New_number=Number+1,
```

```
  summ(New_number,Partial_sum),
  Sum=Number+Partial_sum.

goal
  write("Сума ряду: "),summ(1,Summ).
   /*Результат
   Сума ряду: Summ=55
   1 Solution*/
```

Програма обчислення суми ряду: 2 + 4 + 6 + ... + 14 + 16

```
domains
  number, sum = integer
  predicates
  sum(number, sum)

goal
  write("Сума ряду: "),
  sum(2,Sum).
  clauses
  sum(18,0):-!.
  sum(Number, Sum):-New_number=Number+2,
  sum(New_number,Partial_sum),
  Sum=Number+Partial_sum.

   /*Результат
   Сума ряду: Sum=72
   1 Solution*/
```

Програма обчислення суми ряду: 10 + 9 + 8 + ... + 2 + 1

```
domains
  number, sum = integer

predicates
  sum(number,sum)

goal
  write("Сума ряду:"),
  sum(9,Sum).

clauses
  sum(0,11):-!.
  sum(Number,Sum):-
  New_number=Number-1,
  sum(New_number,Partial_sum),
  Sum=Number+Partial_sum.

   /*Результат
   Сума ряду: Sum=56
   1 Solution*/
```

Програма обчислення суми ряду: 1 + 3 + 5 + ... + 13 + 15

```
domains
```

```
  number, sum = integer

 predicates
  sum(number,sum)

 goal
  write("Сума ряду:"),
  sum(1,Sum).

  clauses
  sum(17,0):-!.
  sum(Number, Sum):-New_number=Number +2,
  sum(New_number, Partial_sum),
  Sum=Number+Partial_sum.

   /*Результат
   Сума ряду: Sum=64
   1 Solution*/
```

Програма, яка друкує таблицю ступенів числа 2, як наведено нижче. Зупинити виконання програми при N=10:

```
 predicates
  count(ulong) - procedure (i)

 clauses
  count(N):-write("N\t2^N"),nl,
  N<=10,!,
  write(N,"\t"),
  Y=exp(N*ln(2)),  % у Пролозі відсутня функція піднесення до степеня
  write(Y),
  NewN = N+1,
  count(NewN),nl.

 goal
  nl,count(1).
```

Приклад трансляції арифметичних виразів над натуральними числами з канонічної (у вигляді структур) форми в постфіксну форму:

```
 predicates
  очікувВигляд(string,string,real)
  розклНаАргум(string,string,string)
  закрити(string,string,string)
  обчислити(string,real,real,real)

 clauses
/* Обираємо дію (знак), розкладаємо вираз на два аргументи та збираємо у потрібному вигляді */
  очікувВигляд(S,S2,R):-
  frontstr(1,S,Z,ST),    % відокремити від рядка 1 символ
  frontchar(ST,'(',SK), % зчитати символ
  str_len(SK,P),         % вивести кількість символів у рядку
```

```
        P1=P-1,
        substring(SK,1,P1,S1),% присвоїти змінній відповідну послідовність символів
        розклНаАргум(S1,AR1,AR2),
        очікувВигляд(AR1,S3,R1),
        очікувВигляд(AR2,S4,R2),
        concat ("(",S3,S5),
        concat (S5,Z,S6),      % об'єднати(конкатенація)
        concat (S6,S4,S7),
        concat (S7,")",S2),
        обчислити(Z,R1,R2,R).
        очікувВигляд(S,S,R):-str_real(S,R).

        /* Підрахунок арифметичних дій */
        обчислити("+",R1,R2,R):-R=R1+R2.  %дія
        обчислити("-",R1,R2,R):-R=R1-R2.
        обчислити("*",R1,R2,R):-R=R1*R2.
        обчислити("/",R1,R2,R):-R=R1/R2.

        /* Пошук коми, що разделяє аргументи з врахуванням парних дужок */
        розклНаАргум(S,AR1,AR2):-frontchar(S,'(',S1),
        закрити(S1,S2,S3),
        concat ("(",S2,S5),
        розклНаАргум(S3,S4,AR2),
        concat (S5,S4,AR1).
        розклНаАргум(S,"",AR2):-frontchar(S,',',AR2).
        розклНаАргум(S,AR1,AR2):-frontstr(1,S,C,S1),
        розклНаАргум(S1,S2,AR2),
        concat (C,S2,AR1).

        /* Пошук дужки, що закриває*/
        закрити(S,AR1,AR2):-frontchar(S,'(',S1),
        закрити(S1,S2,S3),
        конкатенація("(",S2,S5),
        закрити(S3,S4,AR2),
        concat (S5,S4,AR1).
        закрити(S,")",AR2):-frontchar(S,')',AR2).
        закрити(S,AR1,AR2):-frontstr(1,S,C,S1),
        закрити(S1,S2,AR2),
        concat (C,S2,AR1).
        /* ціль програми */

      goal
        очікувВигляд("-(*(+(1,2),3),/(4,5))",S,R),write(S," = ",R),!,nl.
```

Рішення Пролог-програми наступне:

```
(((1+2)*3)-(4/5)) = 8.2
S=(((1+2)*3)-(4/5)), R=8.2
```

Наступний приклад демонструє знаходження визначника матриці порядку 6x6:

```
domains
  rl=real*
```

```
  rll=rl*

 constants
  err=0.0001

 predicates
  det(integer,rll,real)
  det(integer,rll,real,real)
  sub(rll,real,rl,rll)
  sub1(real,real,rl,rl,rl)
  findrow(rll,rll,rl,integer,integer)

 clauses
  det(N,Matr,R):- det(N,Matr,1.0,R).
  det(0,_,R,R):- !.
        det(N,[[A|L]|LL],S,R):- abs(A)>err,!,
        sub(LL,A,L,LL1),N1=N-1,S1=S*A,
        det(N1,LL1,S1,R).
        det(N,LL,S,R):-
        findrow(LL,LL1,L,1,Sign),!,S1=S*Sign,
        det(N,[L|LL1],S1,R).
        det(_,_,_,0.0).
  sub([[B|L]|LL],A,L1,[L2|LL1]):-
  sub1(B,A,L,L1,L2),
  sub(LL,A,L1,LL1).
  sub([],_,_,[]).
        sub1(B,A,[C|L],[C1|L1],[C2|L2]):-
        C2=C-C1*B/A,
        sub1(B,A,L,L1,L2).
        sub1(_,_,[],[],[]).
  findrow([[A|L]|LL],[[A|L]|LL1],L1,S,Sign):- abs(A)<=err,!,S1=-S,
  findrow(LL,LL1,L1,S1,Sign).
  findrow([L|LL],LL,L,S,S).
 goal
det(6,[[1.0,-3.0,4.0,5.0,6.0,7.0],[2.0,1.0,3.0,-6.0,7.0,8.0],[5.0,6.0,7.0,2.0,8.0,9.0],
[5.0,6.0,8.0,2.0,10.0,4.0],[15.0,6.0,18.0,20.0,10.0,11.0],[51.0,60.0,1.0,2.0,-10.0,1.0]],R).
```

Визначник матриці дорівнює

```
R=-1009404
```

Запитання. Завдання

1. У якому випадку доцільно використовувати арифметичні спискові вирази?
2. Які арифметичні оператори включені у Пролог-систему?
3. Які функтори використовуються при обчисленнях у Visual Prolog?
4. Різниця між операторною і структурною формою запису арифметичних виразів.

5. Три властивості оператора для читання арифметичних виразів.
6. Що у Пролог-програмах називають *інфіксними операторами*?
7. Різниця між *інфіксними* та *префіксними* предикатами.
8. Що являє собою *клас пріоритету*?
9. Як виконується предикат з 1-м класом пріоритету?
10. З чого складається арифметичний вираз у Пролог-системі?
11. Які правила діють при узгодженні цілі виду *X=Y* з базою даних?
12. Яка різниця між звичайними предикатами і функціями?
13. Оголошення функцій.
14. Вбудовані функції Пролог?

Тест для самоконтролю знань з розділу 4[F]

1. Основна мета функцій у Пролог-системі?
   а) можливість отримання значень, які повертаються;
   б) можливість отримання недетермінованого рішення;
   в) можливість відрізнити її від рядка символів.
2. Синтаксис оголошення функцій?
   а) звичайна декларація предиката;
   б) на початку декларації редиката розміщується доменне ім'я параметра, що повертається;
   в) після декларації редиката розміщується доменне ім'я параметра, що повертається.
3. З яких компонент може складатись арифметичний вираз у Пролог?
   а) з чисел і арифметичних операторів;
   б) з арифметичних спискових виразів і змінних, які є конкретизовані арифметичним виразо;
   в) з функторі: арифметичних, унарних і функторі перетворення;
   г) всі відповіді вірні.
4. Арифметичні оператори в Пролог – це …?
   а) складання, віднімання, множення і ділення;
   б) остача від ділення цілих чисел та цілочисельне ділення;
   в) всі відповіді вірні.
5. Застосування в обчисленнях Пролог функторів перетворення?
   а) у випадках обчислень при обробці символів;
   б) у випадках обчислень чисел з плаваючою крапкою;
   в) без обмежень;
   г) всі відповіді вірні.
6. Оператори, що записані перед своїми аргументами, називаються …
   а) префіксними операторами;

[F] Усі запитання мають один правильний варіант відповіді.

б) інфіксними операторами;
в) постфіксними операторами.

7. Що собою являє клас пріоритету?
   а) вказують порядок виконання операцій;
   б) визначає, яка операція виконується першою;
   в) ціле число, величина якого залежить від конкретної версії Прологу;
   г) всі відповіді вірні.
8. Що собою представляють аоми +, -, *, /, mod і div?
   а) ц вбудовані предикати Пролог;
   б) функтори, що мають силу тільки в межах арифметичних виразів.
9. У чому різниця між звичайним предикатом і функцією?
   а) значення повертаються через вхідні параметри;
   б) значення, що повертаються, зберігаються в регістрах.
10. У випадку, коли предикат оголошений як функція, яка повертає значення, то …
    а) він може бути викликаний як звичайний предикат Prolog за допомогою додаткового аргументу в якості вихідного аргументу;
    б) він не може бути викликаний як звичайний предикат Prolog за допомогою додаткового аргументу в якості вихідного аргументу.

РОЗДІЛ 5. ОБ’ЄКТИ

У розділі описані об’єкти даних Visual Prolog – це є константи, змінні, символи, числа, атоми, складені об’єкти даних і функтори та багаторівневі об’єкти.

5.1. Прості об’єкти даних

Простий об'єкт даних – це змінна або константа, що ідентифікує об’єкт, який не можна змінювати – аргументи предикатів, а не символьні константи, які визначають в розділі *constants* Пролог-програми (такі як *Pi=3.14* та інші). Простими об’єктами є:

- символ (char);
- число (integer або real);
- атом (symbol або string).

Отже, *змінна* – об’єкт даних. У Пролог змінна може зв’язуватися з будь-яким допустимим аргументом або об’єктом даних. Змінні Прологу локальні, а не глобальні. Так, якщо дві фрази містять змінну (*Ім’я_змінної*), то це дві зовсім різні змінні. Звичайно вони не впливають одна на одну. У випадку, якщо вони співпадуть під час уніфікації, можуть бути пов’язані одна з одною.

Значення константи – це її ім’я. Так числова константа *5* може відповідати тільки числу *5*, а символьна константа *abc* – тільки ідентифікатору *abc*.

Символи мають тип *char*. Печатні символи (згідно кодування ASCII) мають номери 32–127, де стоять цифри (0–9), прописні літери A–Z, рядкові літери a–z, символи пунктуації і спеціальні символи. Літери поза цим діапазоном можуть бути непереносимими з однієї платформи на іншу, особливо, «пропуск» і управляючі символи. Символ константа записується в простих одинарних лапках:

'a' '3' '' '{' 'B' 'A'*

Якщо потрібно записати зворотний слеш «\» або просту одинарну лапку «’», як літерну константу, слід поставити перед нею символ слеш «\» (управляючий escape-символ):

'\\' - backslash '\'' - single quote.

Існує набір символьних констант, які представляють спеціальні функції у тому випадку, якщо їм передує управляючий символ:

'\n' новий рядок (перехід до нового рядка)

'\r' повернення каретки

'\t' горизонтальна табуляція

Символьні константи можуть також бути записані своїм десятковим ASCII-кодом після управляючого символу, наприклад:

'\225' В

'\134' Ж

Числа можуть бути цілими (*integer – 10; -5; 17*) або дійсними (*real – 0.125; -7.22; 7e12*). Дійсні зберігаються в стандартному форматі IEEE і мають значення від від $1*10^{-308}$ до $1*10^{+308}$.

Атоми мають тип ідентифікатор (*symbol*) або рядок (*string*). Відмінність між ними синтаксично не помітна (це головним чином питання машинного уявлення).

Атом передається як аргумент при виклику предиката. До якого домена належить атом – *symbol* або *string* – визначається по тому, як описаний цей аргумент в декларації предиката, наприклад фраза *Студент навчається в «ІП «Стратегія»* оголошена предикатом ідентифікатором *symbol*

```
навчається (symbol, symbol)
```

у розділі clauses має наступний запис

```
навчається(студент, іп_стратегія),
```

а оголошення як рядок

```
навчається (symbol, string)
```

записується у подвійних лапках

```
навчається(студент, "ІП "Стратегія").
```

Visual Prolog автоматично перетворює типи між доменами *string* і *symbol*, тому, можна використовувати атоми *symbol* в доменах *string* і навпаки

```
predicates
студент(string, symbol,word)
clauses
студент (бурлаченко,"П-08-51",100).
```

студент (бачанцева,"П-08-51",101).

Але прийнято вважати, що об’єкт в подвійних лапках належить домену *string*, а об’єкт, що не потребує лапок, домену *symbol*.

Атоми типу *symbol* – це імена, що починаються з рядкової літери, містить різні символи, цифри і знак підкреслення).

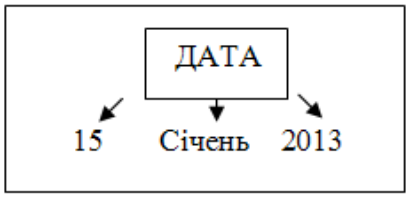
Атоми типу *string* виділяються подвійними лапками і можуть містити будь-яку комбінацію літер, окрім ASCII-нуля (0, бінарний нуль), який позначає кінець рядка атома.

5.2. Складені об’єкти даних

Складені об’єкти даних дозволяють інтерпретувати деякі частини інформації як єдине ціле таким чином, щоб потім можна було легко розділити їх знов.

Оскільки типи *string* та *symbol* взаємозамінні, їх відмінність не істотна. Але імена предикатів і функтори для складених об’єктів повинні відповідати синтаксичним угодам домена *symbol*.

Дата може бути прикладом складеного об’єкта, наприклад дата «15 січня, 2013р.». Інформація, яку містить дата, складається з трьох частин – це день, місяць і рік. Представимо дану інформацію у вигляді деревовидної структури



яку можна реалізувати у Пролог-програмі, оголосивши домен, що містить складений об’єкт *data*:

```
domains
data = data (unsigned,string, unsigned)
% unsigned  – беззнакові кількісні типи

goal
% поставити ціль
D = data (15, "Січень",2013).
```

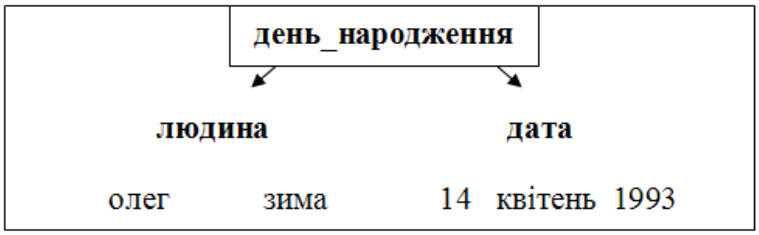
Для цільового твердження знайдене рішення

```
D= data(15, "Січень",2013)
1 Solution
```

Такий запис виглядає як факт Прологу, але це не так – це об’єкт даних, який можна обробляти разом з символами і числами. Він починається з імені, що називається *функтором* (в даному випадку *дата*), за яким слідують три аргументи(число, рядок, число).

Функтор в Visual Prolog не є функцією, як це можна представити в інших мовах програмування, – це просто ім’я, яке визначає вид складеного об’єкту даних і об’єднує разом його аргументи і зовсім не означає, що будуть виконані які-небудь обчислення.

Аргументи складеного об’єкту даних самі можуть бути складеними об’єктами. Наприклад, чий-небудь день народження, як інформація з наступною структурою:



Функтори оголошують у розділі *domains* стандартними доменами. Відповідно синтаксису Пролог, інформація про день народження має такий запис

```
domains
 дата = дата (unsigned,string, unsigned)
 людина= людина(string, string)
 predicates
 %  дата і  людина – це функтори
 день_народження(людина,дата)
clauses
 день_народження(людина("Олег","Зима"),дата(14, "квітень",1993).
```

Функторам можна присвоювати довільні інформативні імена, вимоги до запису такі ж самі як і для предикатів. Наступний приклад демонструє застосування складених структур даних для пошуку фактів, що реалізують можливості Windows для осіб з вадами слуху.

```
domains
 можлWindows=можлWindows(завдання,реалізація)
 завдання=завдання(string)
 реалізація=реалізація(symbol,string)
```

```
predicates
 можлWindows(завдання,реалізація)- nondeterm(o,реалізація(i,o))
 виконати - nondeterm()

clauses
 виконати:-write("**реалізація **"),nl,
   можлWindows(_,реалізація(слуху,Y)),nl,
   write(Y),nl,
   fail.
 виконати.

 можлWindows(завдання("Включення субтитрів"),
              реалізація(слуху," Налагодження Windows")).
 можлWindows(завдання("Включення субтитрів"),
              реалізація(слуху," Налагодження параметрів голосу")).
 можлWindows(завдання("Включення візуального оповіщення"),
              реалізація(слуху," Налагодження Windows")).
 можлWindows(завдання("Включення візуального оповіщення"),
              реалізація(слуху,"Службові програми для спеціальних можливостей")).
 можлWindows(завдання("Створення звукової схеми"),
              реалізація(слуху,"Майстер спеціальних можливостей")).
 можлWindows(завдання("Створення звукової схеми"),
              реалізація(слуху,"Служби корпорації Майкрософт")).
 можлWindows(завдання("Створення звукової схеми"),
              реалізація(слуху,"Спеціальні можливості панелі управління")).
 можлWindows(завдання("Включення режиму високого контрасту"),
              реалізація(зору,"Коригування дозволу екрану")).
 можлWindows(завдання
              ("Коригування налагоджень режиму високої контрастності"),
              реалізація(зору,"Коригування дозволу екрану")).
 можлWindows(завдання("Озвучення перемикання режимів"),
              реалізація(зору,"Налагодження Windows")).
 можлWindows(завдання("Зміна розміру об'єктів і тексту на екрані"),
              реалізація(зору,"Установка степеня зростання")).
 можлWindows(завдання("Зміна розміру об'єктів і тексту на екрані"),
              реалізація(зору,"Коригування дозволу екрану ")).
 можлWindows(завдання("Екранна лупа"),
              реалізація(зору,"Завдання параметрів стеження екранної лупи за
                   діями")).
 можлWindows(завдання("Екранна лупа"),
              реалізація(зору,"Перетворення кольорів вікна екранної лупи")).
 можлWindows(завдання("Екранна лупа"),
              реалізація(зору,"Екранна лупа зі згорнутим у значок д. в.
                   параметрів")).
 можлWindows(завдання("Екранна лупа"),
 реалізація(зору,"Зміна положення вікна екранної лупи")).
 можлWindows(завдання("Екранний диктор"),
              реалізація(зору,"Озвучення подій на екрані")).
 можлWindows(завдання("Екранний диктор"),
              реалізація(зору,"Переміщення покажчика миші до активного
                   елемента")).
 можлWindows(завдання("Екранний диктор"),
```

```
                реалізація(зору,"Читання у голос символів")).
можлWindows(завдання("Екранний диктор"),
                реалізація(зору," Налагодження параметрів голосу")).
можлWindows(завдання("Екранний диктор"),
                реалізація(зору,"Переміщення за допомогою клавіатури і диктора")).
можлWindows(завдання("Налагодження Windows"),
                реалізація(руху,"Зміна конфігурації кнопок миші")).
можлWindows(завдання("Налагодження Windows"),
                реалізація(руху,"Зміна швидкості повтору символу, що вводиться")).
можлWindows(завдання("Налагодження Windows"),
                реалізація(руху,"Управління покажчика з клавіатури")).
можлWindows(завдання("Настройка Windows"),
                реалізація(руху,"Включення альтернативного пристрою
                        вводу")).
можлWindows(завдання("Налагодження Windows"),
                реалізація(руху,"Включення фільтрації вводу")).
можлWindows(завдання("Налагодження Windows"),
                реалізація(руху,"Включення залипання клавіш")).
можлWindows(завдання("Програмне забезпечення"),
                реалізація(руху,"Службові програми")).
можлWindows(завдання("Налагодження Windows"),
                реалізація(руху,"Спеціальні можливості панелі управління")).
goal
 виконати,nl.
```

Результати пошуку Visual Prolog:

```
** Реалізація можливостей Windows для осіб з вадами слуху: **
Налагодження Windows
Налагодження параметрів голосу
Службові програми для спеціальних можливостей
Майстер спеціальних можливостей
Служби корпорації Microsoft
Спеціальні можливості панелі управління
yes
```

Рішення знайдено за правилом, у якому перший функтор *завдання* замінено анонімною змінною. У функторі *реалізація(symbol,string)* перший аргумент є вхідним («*слуху*»). Декларація предиката *можлWindows* виконується у вигляді

```
можлWindows(завдання,реалізація)- nondeterm(o,реалізація(i,o))
```

Предикат містить у своїй структурі функтори *завдання* і *реалізація*, які оголошено у розділі *domains* стандартними доменами. Не слід плутати поняття функтор і власний домен користувача.

5.3. Уніфікація складених об’єктів

Складений об’єкт може бути уніфікований з простою змінною або зі складеним об’єктом (що, можливо, містить змінні як частини у внутрішній

структурі), який йому відповідає. Це означає, що складений об’єкт можна використовувати для того, щоб передавати цілий набір значень як єдиний об’єкт, і потім застосовувати уніфікацію для їх розділення.

Приклад, що демонструє як *дата(15,"січень",2013)* зіставляється зі змінною *X* і змінній *X* привласнює значення *функтора дата(15,"січень",2013)*. Функтор оголошується єдиним вихідним параметром у розділі *predicates* (*nondeterm(o)*).

```
 %******Вільна змінна зіставляється з функтором дата *******
 %Оголосити домен, що містить складений об'єкт дата
domains
 дата = дата(unsigned,string,unsigned)

predicates
 день_нар(дата) -nondeterm(o) %дата - функтор

clauses
 % структура даних
 день_нар(дата(15,"січень",2013)).
 день_нар(дата(15,"листопад",2000)).
 день_нар(дата(15,"жовтень",1993)).

goal
 день_нар(X).
```

Змінна *X* після зв’язування з функтором *дата* у першому факті виводить перше рішення, звільняється і від точки зупинки виконує пошук альтернатив. Результат пошуку у вигляді наступного списку:

```
X=дата(15,"січень",2013)
X=дата(15,"листопад",2000)
X=дата(15,"жовтень",1993)
3 Solutions
```

У складених структурах звичайні змінні можуть зіставлятися з аргументами функторів, наприклад *D=15, M="січень"* і *Y=2013*. У розділі *predicates* оголошується три вихідних параметри - *nondeterm(дата(o,o,o))*.

```
 %***Вільні змінні зіставляються з аргументами функтора ***
domains
 дата = дата(unsigned,string,unsigned)

predicates
 день_нар(дата) -nondeterm(дата(o,o,o))  %дата - функтор

clauses
 день_нар(дата(15,"січень",2013)).
 день_нар(дата(15,"листопад",2000)).
```

```
  день_нар(дата(15,"жовтень",1993)).

goal
  день_нар(дата(D,M,Y)).
```

Результат пошуку:

```
D=15, M=січень, Y=2013
D=15, M=листопад, Y=2000
D=15, M=жовтень, Y=1993
3 Solutions
```

Для цільового твердження *run* побудоване правило, у якому змінна *Дата* зіставляється з функтором *дата*

```
  %*** У правилі вільна змінна зіставляється з функтором ***
domains
  дата = дата(unsigned,string,unsigned)

predicates
  день_нар(дата) -nondeterm(o)  %дата - функтор
  run()

clauses
  % Структура даних
  день_нар(дата(15,"січень",2013)).
  день_нар(дата(15,"листопад",2000)).
  день_нар(дата(15,"жовтень",1993)).
  run:-день_нар(Дата),write(Дата),nl.

goal
  run.
```

У даному випадку Пролог знаходить одне рішення і не виконує пошуку альтернативних рішень

```
дата(15,"січень",2013)
yes
```

Якщо останньою підціллю у правило додати завжди неуспішний предикат *fail*,

```
run:-день_нар(Дата),write(Дата),nl,fail.
```

то Пролог знайде множинні рішення для цілі *run*

```
дата(15,"січень",2013)
дата(15,"листопад",2000)
дата(15,"жовтень",1993)
no
```

але програма закінчиться неуспішно, тому що відсутній факт *run* для цілі у розділі *clauses*. Якщо такий факт існує

```
run:-день_нар(Дата),write(Дата),nl,fail.
run.
```

то Пролог завершить пошук успішно *yes*.

У правилах і цілях Пролог-програм для складених структур можна застосовувати анонімні змінні для ігнорування деяких аргументів функтора

```
 %******використання анонімних змінних **********
domains
 дата = дата(unsigned,string,unsigned)

predicates
 день_нар(дата) -nondeterm(дата(o,o,o))  %дата - функтор

clauses
 день_нар(дата(15,"січень",2013)).
 день_нар(дата(15,"листопад",2000)).
 день_нар(дата(15,"жовтень",1993)).

goal
 день_нар(дата(D,M,_)).
```

Під час пошуку не беруться до уваги об’єкти, що визначають рік у складеній структурі

```
D=15, M=січень
D=15, M=листопад
D=15, M=жовтень
3 Solutions
```

Аргументи функтора зіставляються зі звичайними змінними у тілі правила

```
 %*** змінні у правилі ***
domains
 дата = дата(unsigned,string,unsigned)

predicates
 день_нар(дата) %-nondeterm(дата(o,o,o))  %дата - функтор
 run()

clauses
 день_нар(дата(15,"01",2013)).
 день_нар(дата(15,"09",2000)).
 день_нар(дата(15,"05",1993)).
 run:-день_нар(дата(D,M,Y)),write(D,"/",M,"/",Y),nl,fail.
 run.

goal
 run.
```

Результат пошуку

```
15/01/2013
```

```
15/09/2000
15/05/1993    yes
```

5.4. Використання знаку рівності для уніфікації складених об’єктів

Visual Prolog здійснює уніфікацію у двох випадках:

- коли ціль зіставляється із заголовком правила;
- через знак рівності (=), який є *інфіксним* предикатом (предикатом, який розташований між своїми аргументами, а не перед ними).

Фактично, Visual Prolog виконує операцію привласнення для уніфікації об’єктів по різні сторони знаку рівності. Ця властивість корисна для знаходження значень аргументів складеного об’єкта.

Наприклад, Пролог перевіряє, чи співпадають прізвища у двох людей, і потім дає другій людині ту ж адресу, що і у першої.

```
domains
 person = person(name,address)
 name = name(first,last)
 address = addr(street,city,state)
 street = street(number,street_name)
 city,state,street_name = string
 first,last = string
 number = integer

goal
 PI=person(name(jim,mos),addr(street(5,"1stst"),igo,"CA")),
 PI = person(name(_,mos),Address)
 P2=person(name(jane,mos),Address), write("PI=",PI),nl
 write("P2=",P2),nl.
```

Складені об’єкти можуть розглядатися у фразах Прологу як єдині об’єкти, що суттєво спрощує написання програм.

Розглянемо, факт:

```
має(олег, книга("Звідси у вічність","Джеймс Джонсон")).
```

у якому стверджується, що у Олега є книга «Звідси у вічність» написана Джеймсом Джонсом. Аналогічно можна записати:

```
Олег має авто «Ford».
має(олег, авто(форд)).
```

Складеними об’єктами в цих двох прикладах є:

книга("Звідси у вічність", "Джеймс Джонсон") і *авто(форд)*

Якщо натомість описати тільки два факти:

```
має("Звідси у вічність","Джеймс Джонсон")
має(форд)
```

то не можна було б визначити, чи є *форд* назвою книги або назвою *авто*. З іншого боку, можна використовувати перший компонент складеного об’єкту – функтор для розпізнавання різних об’єктів. Цей приклад використовує функтори книга і авто для вказівки різниці між об’єктами.

Складені об’єкти складаються з функтора і об’єктів, що належать цьому функтору.

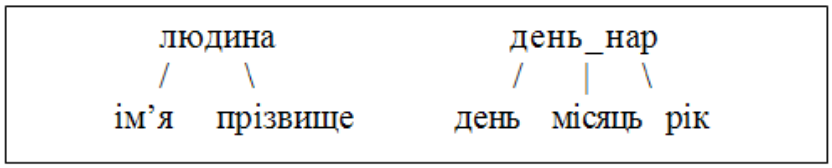
Важлива особливість складених об’єктів полягає в тому, що вони дозволяють легко передавати групи величин, як один аргумент.

Прикладом може бути ведення телефонної бази даних. Припустимо, потрібно включити в базу даних дні народження друзів. Для цього довелося б написати програму, частина якої приведена нижче:

```
predicates
список(symbol Ім’я, symbol Прізвище, symbol Телефон,
       integer День, symbol Місяць, integer Рік)

clauses
список(олег, зима, 53575, 3, січень, 1995).
список(ігор, лига, 27558, 15, квітень, 1992).
```

У факті список з шести аргументів, п’ять з них можуть бути розбиті на два складені об’єкти.



Може виявитися кориснішим представляти факти так, щоб вони відображали ці складені об’єкти даних.

Повернувшись на крок назад, ясно, що людина – це відношення, а ім’я і прізвище – об’єкти. Також, *день_нар* – це відношення між трьома аргументами: днем, місяцем і роком. У представленні Прологу вони можуть бути записані таким чином:

```
людина (ім'я, прізвище)
день_нар (днь, місяць і рік)
```

Тепер можна переписати свою маленьку базу даних з включенням цих складених об’єктів як частини бази даних.

```
domains
персона = людина (symbol Ім'я, symbol Прізвище)
день_нар= дата(integer День,symbol Місяць,integer Рік)
телефон = symbol

predicates
список (персона, телефон, день_нар)
clauses
список(людина(олег,зима),53575, дата(3,січень, 1995)).
список(людина(ігор,лига),27558, дата(15,квітень, 1992)).
```

У цю програму введені два визначення складених доменів.

Предикат *список* тепер містить три аргументи, що відрізняється від шести – в попередньому прикладі. Іноді розбиття даних в складеному об’єкті робить яснішої логіку програми і може допомогти в обробці даних.

Приклад.

```
    domains
    %3 рівень
вид_навантаження = вид_навантаження(назва_навантаження, форма_роботи)
    %2 рівень
види_робіт = види_робіт(назва_виду_роботи, вид_навантаження)
    %1 рівень
назва, форма_роботи, назва_виду_роботи, назва_навантаження = string

    predicates
предмет(назва,види_робіт)-nondeterm(o,види_робіт(o,вид_навантаження(o,o)))
    правило()-nondeterm()

    clauses
     предмет("Логічне програмування",види_робіт("Аудиторна",
            вид_навантаження("Практична", "Лабораторна"))).

     предмет("Менеджмент ", види_робіт("Домашня",
            вид_навантаження("Самостійна", "Реферат"))).

     предмет("США ", види_робіт("Аудиторна",
            вид_навантаження("Практична", "Лабораторна"))).

     правило():-предмет(Назва_предмету, види_робіт(Вид_роботи,
            вид_навантаження(Назва_навантаження, Форма))),
            write(Назва_предмету, "\n\t", Вид_роботи, "\n\t\t",
            Назва_навантаження, "\n\t\t\t", Форма), nl,fail.
    равило.

    goal
    правило().
```

```
/*
Логічне програмування
   Аудиторна
         Практична
               Лабораторна
Менеджмент
   Домашня
         Самостійна
               Реферат
США
   Аудиторна
         Практична
               Лабораторна
yes
*/
```

5.5. Оголошення складених доменів

Як визначаються складені домени? Після компіляції програми, яка містить наступні відношення:

```
має(олег, книга("Звідси у вічність","Джеймс Джонсон")).
має(олег, авто(форд)).
```

слід послати системі запит в наступному вигляді:

```
має(олег, X).
```

Змінна *X* може бути пов’язана з різними типами об'єктів: книга, авто та ін., які може визначити користувач.

Тепер не можливо використовувати старе визначення предиката *має*:

```
має(symbol, symbol)
```

Другий елемент більш не є об’єктом типу *symbol*. Натомість можна дати нове визначення цього предиката:

```
має(name, власність).
```

Домен власність в розділі domains можна описати так:

```
domains
власність = книга(назва,автор); авто(марка)
% власність - це книга або авто
назва,автор,марка=symbol
```

Крапка з комою читається як «*або*».

Домени *назва*, *автор* і *ім’я* мають стандартний тип *symbol*.

До визначення домена легко можуть бути додані інші варіанти. Наприклад, власність може також включати *човен*, *будинок*, *чекову книжку* тощо.

Човен можна визначити функтором без аргументів.

Човен(). або *човен.*

Можна включити платіжний *баланс*, як частину *чекової_книжки*. Визначення домена власність розшириться до:

```
domains
власність=книга(назва,автор); авто(марка); човен; чек_книга(баланс)
назва, автор, марка = symbol
баланс = real
```

Складені об’єкти з домена власність можуть використовуватися у фактах, які визначають предикат має:

```
predicates
  має(ім’я,власність)

clauses
  має(олег, книга("Звідси у вічність","Джеймс Джонсон")).
  має(олег, авто(форд)).
  має(олег, човен).
  має(олег, чек_книгу(1000)).

goal
  має (олег, Що).
```

Visual Prolog виведе рішення:

```
Що= книга("Звідси у вічність","Джеймс Джонсон")
Що= авто(форд)
Що= човен
Що= чек_книгу(1000)
4 Solutions
```

Приклад.

```
domains
список=список(факультет,напрям)
факультет=факультет(string)
напрям=напрям(string)
інф=інф(напрям,група)
група=група(string,піб)
піб=піб(string,string,string,string)

predicates
структура - nondeterm()
список(факультет,напрям)- nondeterm(факультет(о),напрям(о))
інф(напрям,група)- nondeterm (і,група(о,о)),(і,група(і,піб(о,о,о,о)))

clauses
структура:-
    write("Список студентів факультету за напрямками"),nl,
```

```
    список(факультет(F),
    напрям(N)),
    write("\t Факультет ",F),nl,
    write("\t\t напрям ",N),nl,
    інф(напрям(N),група(G,_)),
    write("\t\t\t Група ",G,":"),nl,
    інф(напрям(N),група(G,піб(P,I,S,T))),
    write("\t\t\t\t Студент ",P," ",I," ",S," ",T),nl,
    fail.

структура.
список(факультет("Управління"),напрям("Комп'ютерні науки")).
список(факультет("Управління"),напрям("Економічна кібернетика")).
список(факультет("Управління"),напрям("Менеджмент")).
список(факультет("Управління"),напрям("Економіка підприємств")).


інф(напрям("Комп'ютерні науки"),група("П-09-51",
            піб("Прізвище1","Ім'я1",чол,"м.Жовті Води"))).
інф(напрям("Економіка підприємств"),група("Е-09-51",
            піб("Прізвище2","Ім'я2",чол,"м.Жовті Води"))).
інф(напрям("Економічна кібернетика"),група("С-09-51",
            піб("Прізвище3","Ім'я3",чол,"м.Жовті Води"))).
інф(напрям("Менеджмент"),група("М-09-51",
            піб("Прізвище4","Ім'я4",чол,"м.Жовті Води"))).
інф(напрям("Менеджмент"),група("М-09-51",
            піб("Прізвище5","Ім'я5",чол,"м.Жовті Води"))).
інф(напрям("Комп'ютерні науки"),група("П-09-51",
            піб("Прізвище6","Ім'я6",чол,"м.Жовті Води"))).
інф(напрям("Економічна кібернетика"),група("С-09-51",
            піб("Прізвище7","Ім'я7",чол,"м.Жовті Води"))).
інф(напрям("Комп'ютерні науки"),група("П-09-51",
            піб("Прізвище8","Ім'я8",чол,"м.Жовті Води"))).

goal
структура.

/*Список студентів факультету за напрямами
    Факультет Управління
        напрям Комп'ютерні науки
            Група П-09-51:
                Студент Прізвище1 Ім'я1 чол м.Жовті Води
                Студент Прізвище6 Ім'я6 чол м.Жовті Води
                Студент Прізвище8 Ім'я8 чол м.Жовті Води
            Група П-09-51:
                Студент Прізвище1 Ім'я1 чол м.Жовті Води
                Студент Прізвище6 Ім'я6 чол м.Жовті Води
                Студент Прізвище8 Ім'я8 чол м.Жовті Води
            Група П-09-51:
                Студент Прізвище1 Ім'я1 чол м.Жовті Води
                Студент Прізвище6 Ім'я6 чол м.Жовті Води
                Студент Прізвище8 Ім'я8 чол м.Жовті Води
```

```
        Факультет Управління
            напрям Економічна кібернетика
                Група С-09-51:
                    Студент Прізвище3 Ім'я3 чол м.Жовті Води
                    Студент Прізвище7 Ім'я7 чол м.Жовті Води
                Група С-09-51:
                    Студент Прізвище3 Ім'я3 чол м.Жовті Води
                    Студент Прізвище7 Ім'я7 чол м.Жовті Води
        Факультет Управління
            напрям Менеджмент
                Група М-09-51:
                    Студент Прізвище4 Ім'я4 чол м.Жовті Води
                    Студент Прізвище5 Ім'я5 чол м.Жовті Води
                Група М-09-51:
                    Студент Прізвище4 Ім'я4 чол м.Жовті Води
                    Студент Прізвище5 Ім'я5 чол м.Жовті Води
        Факультет Управління
            напрям Економіка підприємств
                Група Е-09-51:
                    Студент Прізвище2 Ім'я2 чол м.Жовті Води
    yes*/
```

5.6. Декларація доменів для складених об’єктів

Синтаксис запису наступний

домен = альтернатива1(Ім’я Коментар,...); альтернатива2(Ім’я Коментар,...); ...

Тут *альтернативаN* - допустимі (але різні) функтори.

Запис (*Ім’я, Ім’я*, ...) – представляє список імен доменів, які оголошені десь в програмі або є стандартними типами доменів (такими як *symbol*, *integer*, *real* і ін.).

Необов’язкові параметри *Коментар* можуть використовуватися для коментаря імен аргументів, вони ігноруватимуться компілятором. *Альтернативи* розділяються крапкою з комою. Кожна альтернатива складається з функтора і, можливо, списку доменів відповідних аргументів. Якщо функтор не має аргументів, його можна записати в програмі *альтернативаN* або *альтернативаN( )*. Частіше використовується перший варіант синтаксису.

5.7. Багаторівневі складені об’єкти

Visual Prolog дозволяє конструювати складені об’єкти на декількох рівнях. Наприклад, в:

```
    книга("Гидке каченя","Андерсен")
```

замість прізвища автора можна використовувати нову структуру, яка описує автора детальніше, включаючи ім’я і прізвище. Викликаючи функтор для нового об’єкту автор (автор), ви можете змінити опис книги:

```
книга("Гидке каченя",автор("Ганс Крістіан","Андерсен"))
```

У старому визначенні домена

```
книга (назва, автор)
```

другим аргументом функтора *книга* був *автор*. Але старе визначення

```
автор = symbol
```

може включати тільки одне ім’я, а цього вже недостатньо. Визначимо тепер автор, як складений об’єкт, що складається з імені і прізвища автора.

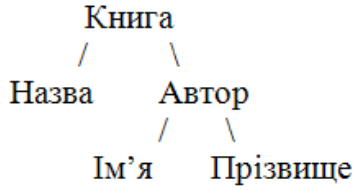
Це досягається за допомогою декларації наступного домена:

```
автор = автор(імя,прізвище)
```

що приводить до наступних визначень:

```
domains
видання  = книга(назва, автор);    % Перший рівень
автор = автор(імя, прізвище)       % Другий рівень
назва, імя, прізвище = symbol      % Третій рівень
```

При використанні складених об’єктів з багатьма рівнями часто допомагає «дерево»:



Декларація домена оголошує тільки один рівень дерева, а не ціле дерево. Отже, книга не може бути описана такою фразою:

```
domains
книга=книга(назва,автор(імя,прізвище))%неправильно
```

5.8. Визначення складених змішаних доменів

Слід розглянути різні типи визначень доменів, які можна додавати до програм. Ці оголошення дозволять використовувати предикати, які мають можливість:

- одержувати аргумент більш ніж одного типа;
- одержувати різну кількість аргументів, всі різних вказаних типів;

- одержувати різну кількість аргументів, деякі з яких можуть бути більш ніж одного з можливих типів.

Для того, щоб дозволити предикатам Visual Prolog одержувати аргументи, які містять інформацію різних множинних типів, потрібно додати опис функтора.

У наступному прикладі фраза ***ваш_вік(вік)*** одержує аргумент типу ***вік***, який може мати тип *string*, *real* або *integer*.

```
 % аргументи містять дані різних типів
domains
 вік=i(integer); r(real); s(string)
 %вік=integer;real;string - така декларація недопустима

predicates
 ваш_вік(вік)

clauses
 ваш_вік(i(Вік)):-write(Вік),nl.
 ваш_вік(r(Вік)):-write(Вік),nl.
 ваш_вік(s(Вік)):-write(Вік),nl.

goal
ваш_вік(s(тридцять)).  %  або
%ваш_вік(r(30.5)).
%ваш_вік(i(30)).
```

Запитання. Завдання

1. Які типи об’єктів існують у Visual Prolog?
2. До якого типу об’єктів відносять аргументи предикатів?
3. Які символьні константи, представляють спеціальні функції, якщо їм передує управляючий символ?
4. Що таке «атом»?
5. Пояснити термін «функтор».
6. Розділ, у якому оголошують функтор?
7. Способи уніфікації складених об’єктів.
8. За яких умов Visual Prolog здійснює уніфікацію об’єктів?
9. Що взодить до складених об’єктів?
10. Як визначаються складені домени?
11. Вимоги до декларації складених об’єктів даних.

12. Чи може бути оголошеним складений об’єкт більш ніж одного типу?

Тест для самоконтролю знань з розділу 5[Ғ]

1. Простий об’єкт даних – це ...
   а) константа, що ідентифікує об’єкт, який не можна змінювати;
   б) змінна або константа, що ідентифікує об’єкт, який не можна змінювати;
   в) символьна константа, що ідентифікує об’єкт, який не можна змінювати;
   г) змінна або символьна константа, що ідентифікує об’єкт, який не можна змінювати.
2. Простим об’єктом Visual Prolog є:
   а) символ;
   б) число;
   в) атом;
   г) всі відповіді вірні.
3. Якому типу відповідають атоми у Пролозі?
   а) символ (char);
   б) ідентифікатор (symbol) або рядок (string);
   в) число (integer або real);
   г) всі відповіді вірні.
4. Складені об’єкти даних - це ...
   а) об’єкти, що дозволяють інтерпретувати деякі частини інформації як єдине ціле;
   б) ідентифікатор об’єкти, що являють собою предикат N-ї арності з аргументами;
   в) об’єкти, що використовуються у складених підцілях;
   г) всі відповіді вірні.
5. Функтор – це ...?
   а) тип даних;
   б) ім’я предиката N-ї арності з аргументами;
   в) ім’я, яке визначає вид складеного об’єкту даних і об’єднує разом його аргументи.
6. Аргументами складеного об’єкту є ...
   а) об’єкти стандартного типу;
   б) об’єкти користувацьких типів;
   в) складені об’єкти;
   г) всі відповіді вірні.
7. Як уніфікується складений об’єкт?
   а) з простою змінною;
   б) зі складеним об’єктом;
   в) з анонімною змінною;
   г) всі відповіді вірні.
8. Значення знака рівності «=» у Пролозі?

[Ғ] Усі запитання мають один правильний варіант відповіді.

а) присвоювання значення;
б) інфіксний предикат, який розташований між своїми аргументами;
в) операція привласнення.

9. Функтор – ...
а) це атом;
б) користувацький тип даних;
в) це 1-й компонент складеного об’єкту, призначений для їх розпізнавання;
г) це будь-який компонент складеного об’єкту, призначений для їх розпізнавання.

10. Декларація доменів для складених об’єктів?
а) функтор1(Ім’я Коментар,...); функтор2(Ім’я Коментар,...); ...
б) функтор1(Ім’я Коментар,...), функтор2(Ім’я Коментар,...), ...
в) ім’я = функтор1(Ім’я Коментар,...);
функтор2(Ім’я Коментар,...); ...
г) ім’я = функтор1(Ім’я Коментар,...), функтор2(Ім’я Коментар,...);,...

11. Яка існує можливість оголошення складених змішаних доменів?
а) одержувати різну кількість аргументів, всі різних вказаних типів;
б) одержувати різну кількість аргументів, деякі з яких можуть бути більш ніж одного з можливих типів;
в) одержувати аргумент більш ніж одного типу;
г) всі відповіді вірні.

12. Аргументи функтора можуть одночасно містити дані типів …
а) integer та symbol;
б) integer та real;
в) real, integer, string
г) будь-які, оголошені в domains.

13. Оголошений об’єкт
інф(напрям,група)- nondeterm (і,група(о,о)),(і,група(і,піб(о,о,о,о))) а)
а) оголошений простий об’єкт;
б) об’єкт має дворівневу структуру;
в) об’єкт має трирівневу структуру;
г) об’єкт має чотирирівневу структуру.

14. Вказати правильний спосіб оголошення функторів?
а) *видання = книга(назва, автор); автор = автор(імя, прізвище); назва, імя, прізвище = symbol*
б) *видання = книга(назва, автор), автор = автор(імя, прізвище)*
назва, імя, прізвище = symbol
в) *видання = книга(назва, автор)*
автор = автор(імя, прізвище)
назва, імя, прізвище = symbol
г) всі відповіді вірні.

15. Факт *інф(напрям("Комп'ютерні науки"),група("П-09-51",*
піб("Прізвище1","Ім'я1",чол,"м.Жовті Води"))).
    а) оголошений як простий об’єкт;
    б) оголошений як складений об’єкт дворівневої структури;
    в) оголошений як складений об’єкт трирівневої структури;
    г) оголошений як складений об’єкт чотирирівневої структури.
16. Вказати правильний спосіб оголошення доменів складених об’єктів?
    а) вік=i(integer); r(real); s(string)
    б) вік=i(integer), r(real), s(string)
    в) вік=i(integer)
        r(real)
        s(string)
    г) всі відповіді вірні.

РОЗДІЛ 6. ПОВТОР І РЕКУРСІЯ

Visual Prolog може виражати повтор як в процедурах, так і в структурах даних. Він дозволяє створювати структури даних, що повторюються, розмір яких під час створення не відомий.

Класичний Пролог не має явних конструкцій операторів циклу *For Next* , *Do While* – у його синтаксисі не існує прямого способу виразу повтору. Мова Visual Prolog забезпечує тільки два види повторення:

- відкат, за допомогою якого здійснюється пошук багатьох рішень в одному запиті;
- і рекурсію, в якій процедура викликає сама себе.

Пролог розпізнає спеціальний випадок так званої *хвостової рекурсії* і компілює її в *оптимізовану ітераційну петлю*. Хоча програмна логіка і виражається рекурсивно, скомпільований код такий же ефективний як у C або C#.

Рекурсія у багатьох випадках більш логічна і менше допускає помилок, ніж цикли, що використовуються імперативними мовами програмування.

Коли виконується процедура пошуку з поверненням (відкат), відбувається пошук іншого рішення цільового твердження. Це здійснюється шляхом повернення до останньої з перевірених підцілей, що має альтернативне рішення, з використанням наступної альтернативи цієї підцілі і нової спроби руху вперед.

Пошук з поверненням – це могутній і ефективний засіб з погляду економії пам’яті, але змінні обновляються після кожної ітерації, тому їх значення втрачаються. Рекурсія дозволяє змінювати значення змінних, але вона неефективна з погляду використованого обсягу пам’яті.

Visual Prolog може оптимізувати хвостові рекурсії, що знімає вимоги рекурсії до об’єму пам’яті.

6.1. Використання відкату з петлями

Пошук з поверненням є ефективним способом визначити всі можливі рішення цільового твердження. Але, навіть якщо задача не має багато рішень,

можна використовувати пошук з поверненням для виконання *ітерацій* (кроків). Для цього слід просто визначити предикат з двома фразами:

```
repeat.
repeat : - repeat.
```

Такий прийом демонструє створення структури управління Прологу, яка породжує нескінченну множину рішень, наприклад:

```
predicates
  місто(symbol) - nondeterm (o)
  печатати - nondeterm ()
  repit

clauses
  місто("Київ").
  місто("Дніпропетровськ").
  місто("Жовті Води").
  місто("Львів").
  місто("Харків").
  repit.              % предикат нічого не виконує і ставить точку відкату
  repit:-repit.
  печатати:- repit, місто(X), write(X), nl, fail.
  печатати.

goal
  печатати.
```

Ціль предиката *repeat* – виконувати нескінченний пошук з поверненням (нескінченну кількість відкатів). Зупинити нескінченний друк результатів можна стандартною комбінацією клавіш [*Ctrl+Break*].

Наступний приклад демонструє використання *repeat* для збереження введених символів та їх друку до тих пір, поки користувач не натисне клавішу [*Enter*]:

```
predicates
  repeat
  typewriter

clauses
  repeat.
  repeat :- repeat.
  typewriter :- repeat,
  readchar(X),  % ввести символ, зчитати, привласнити його значення X
  write(X),!.     % або привласнити символ повернення каретки X = '\r' (Enter)

goal
  typewriter(),nl.
```

Програма показує, як працює предикат *repeat*. Правило *typewriter:-* описує процес прийому символів з клавіатури і відображення їх на екрані. Правило *typewriter* працює таким чином:

- виконує *repeat*, який ставить точку відкату і більше нічого не виконує;
- привласнює змінній *X* значення символу;
- відображає значення змінної *X*.
- виконується завершення пошуку. Оскільки ні *write*, ні *readchar* не мають альтернатив, постійно відбувається повернення до *repeat*, який завжди має альтернативні рішення.

Тепер обробка знову просувається вперед, зчитується і відображається наступний символ, відсікання припиняє пошук альтернатив.

Змінна *X* втрачає своє значення після відкату в позицію перед викликом предиката *readchar(X)*, який зв'язує змінну *X*.

Такий тип звільнення змінної є істотним, коли пошук з поверненням застосовується для визначення альтернативних рішень цільового твердження, але не ефективний при використовуванні пошуку з поверненням в інших цілях. Проблема в тому, що хоча пошук з поверненням і може повторювати операції скільки завгодно разів, він не здатний запам'ятати і передати що-небудь з одного повторення для іншого.

Всі змінні втрачають свої значення, коли обробка відкочується в позицію, попередню тим викликам предикатів, які ці значення встановлювали.

Не існує простого способу збереження значень лічильника числа повторень петлі або будь-яких інших значень, які були обчисленими при виконанні циклів повторення.

6.2. Рекурсивні процедури

Інший спосіб організації повторень – рекурсія.

Рекурсивна процедура – це процедура, яка викликає сама себе.

Відкат дає можливість отримати багато рішень в одному питанні до програми, а рекурсія дозволяє використовувати в процесі визначення предиката його самого.

Рекурсія є основним прийомом програмування в Пролозі. Більш того, Пролог дозволяє визначати рекурсивні структури даних.

В рекурсивній процедурі немає проблеми запам'ятовування результатів її виконання, тому що будь-які обчислені значення можна передавати з одного виклику в інший як аргументи предиката, що рекурсивно викликається.

Рекурсія має три основні переваги:

- може виражати алгоритми, які не можна зручно виразити ніяким іншим чином;
- логічно простіше методу ітерацій;
- широко використовується в обробці списків.

Рекурсія — хороший спосіб для опису задач, що містять в собі підзадачу такого ж типу. Як приклад, пошук в дереві, коли дерево складається з дрібніших дерев, та рекурсивне сортування. Для сортування списку, він розділяється на частини, частини сортуються і потім об'єднуються разом.

Логіка рекурсії проста для здійснення. Для усвідомлення її дій слід розглянути наступний класичний приклад, що описує родинні зв'язки людей через відношення «бути батьком».

Предикат *батько* має два аргументи: перший – ім'я батька, другий – ім'я дитини. Завданням є створення відношення «бути батьком», використовуючи предикат *батько*.

Для того, щоб одна людина була предком іншої людини, потрібно, щоб він або був його батьком, або був батьком іншого його предка. Мовою Пролога ці ідеї записують таким чином:

```
предок(Предок, Нащадок):-        % предком є батько
батько(Предок, Нащадок).
предок(Предок, Нащадок):-        %предком є батько предка
батько(Предок, Людина),
предок(Людина, Нащадок).
```

Відношення *предок* є *транзитивним замиканням* відношення *батько*, тобто це якнайменше відношення, яке включає відношення *батько* і володіє властивістю *транзитивності* (відношення називається транзитивним, якщо для будь-яких пар (A, B) і (B, C), що знаходиться в цьому відношенні, пара (A, C) також знаходиться в цьому відношенні).

Очевидно, що відношення *предок* містить відношення *батько*. Це витікає з першої фрази, в якій записано, що всякий батько є предком. Друга фраза дає транзитивність.

По аналогії з математичною індукцією, на яку рекурсія трохи схожа, будь-яка рекурсивна процедура повинна включати *базис* і *крок рекурсії*.

Базис рекурсії – це фраза, що визначає якусь початкову ситуацію або ситуацію у момент припинення.

Як правило, в цій фразі записується якийсь найпростіший випадок, при якому відповідь отримують відразу навіть без використовування рекурсії. Так, в приведеній вище процедурі, що описує предикат *предок*, базисом рекурсії є перше правило, в якому визначено, що найближчими предками людини є його батьки. Ця фраза часто містить умову, при виконанні якої відбувається вихід з рекурсії, або відсікання.

Крок рекурсії – це правило, в тілі якого обов'язково міститься, як підцілі, виклик *визначального* предиката.

Щоб уникнути зациклення, *визначальний предикат* повинен викликатися не від тих же параметрів, які вказані в заголовку правила. Параметри повинні змінюватися на кожному кроці так, щоб у результаті або спрацював базис рекурсії, або умова виходу з рекурсії, розміщена в самому правилі.

Формат правила, що реалізує крок рекурсії:

```
ім'я_визначального_предиката:-
[підцілі],
[умова виходу з рекурсії],
[підцілі],
ім'я_визначального_предиката,
[підцілі].
```

В деяких ситуаціях фрази, що реалізовують базис рекурсії, і фрази, що описують крок рекурсії, можуть містити умови. Як правило, це буває в складних випадках, наприклад, коли виконувані в процесі реалізації кроку рекурсії дії залежать від виконання або невиконання якої-небудь умови як наведено у наступному прикладі. Результатом програми є таблиця ступенів числа 2, тобто 2^N, де N – змінюється від 1 до 10 з кроком 1:

```
N  2^N
_____________
1  1
2  4
3  8
4  16
5  32
6  64
7  128
8  256
9  512
10          1024
```

Примітка. У Visual Prolog відсутня функція обчислення ступеня числа X^N. Для обчислень слід використовувати експоненту і натуральний логарифм числа exp(N*ln(X)).

Текст програми:

```
predicates
  counter(ulong, ulong) - procedure (i, i)

clauses
  counter(N, NbyN):-
  write("\r",N),
        % число, що відповідає 2^N
  write("\t",NbyN),
        % підвищуємо ступінь на 1
  NewN = N+1,
        % перевіряємо, щоб значення ступеня не перевищувало 10
  NewN <= 10,
        % присвоюємо NewNbyN значення попереднього NewNbyN*2
  NewNbyN = exp(NewN*ln(2)),
        % визиваємо рекурсивно функцію знаходження наступного значення
  counter(NewN, NewNbyN).

goal
  nl,
  % вивід заголовку
  write("N"),
  write("\t"),
  write("2^N"),
```

```
write("\r_____________\r"),
        % пошук рішення, починаючи з 1
counter(1, 1).
```

У наступному прикладі наведено простий випадок рекурсії, коли рекурсивна процедура має один базис і один крок рекурсії. Створимо предикат, який обчислюватиме по натуральному числу його факторіал. Ця задача допускає рекурсивне рішення на багатьох мовах програмування, а також має рекурсивний математичний опис знаходження факторіала числа N:

```
1!=1                    % факторіал одиниці дорівнює одиниці
N!=(N-1)!*N             /* для того, щоб обчислити факторіал
                        деякого числа, потрібно обчислити
                        факторіал числа на одиницю меншого
                        і помножити його на початкове число */
```

Цей підхід означає наступне: наприклад, щоб знайти факторіал 3, потрібно знайти факторіал 2, а щоб знайти факторіал 2, потрібно обчислити факторіал 1.

Можна знайти факторіал 1 без звернення до інших факторіалів, тому повторення не почнуться. Якщо факторіал є 1, то слід помножити його на 2, щоб отримати факторіал 2, а потім помножити отримане на 3, щоб отримати факторіал 3. В Visual Prolog це записується так:

```
factorial(1, 1):- !.
factorial(X, FactX):-
                    Y = X -1,
                    factorial(Y, FactY),
                    FactX = X * FactY.
```

Рекурсивна програма обчислення факторіалів представлено наступним кодом із застосуванням звичайної (не хвостової) рекурсії:

```
predicates
  factorial(unsigned, real)

clauses
  factorial(1,1):- !.
  factorial(X, FactX):-
  Y=X-1,
  factorial(Y, FactY),
  FactX = X * FactY.

goal
  X=3,
  factorial(X, FactX).
```

Комп'ютер виконує предикат *factorial* в середині обробки предиката *factorial.* Яким чином це відбувається? Якщо викликати *factorial* з *X=3*, то предикат *factorial* звернеться до себе з *X=2*. Чи буде тоді *X* мати два значення, чи друге значення заміняє перше, чи є інші варіанти?

Пояснити все можна тим, що комп'ютер створює нову копію предиката *factorial* таким чином, що *factorial* стає здатним викликати сам себе як повністю самостійну процедуру. При цьому, код виконання не копіюватиметься, але всі аргументи і проміжні змінні копіюються.

Інформація зберігається у стековому фреймі в області пам'яті, який створюється кожного разу при виклику правила. Коли виконання правила завершується, зайнята його стековим фреймом пам'ять звільняється (якщо це не недетермінований відкат), і виконання продовжується у стековому фреймі батьківського правила.

Наведений вище рекурсивний приклад обчислення факторіала описує нескінченну множину різних обчислень за допомогою лише двох фраз, значно спрощує їх розуміння. Крім того, правильність кожної фрази може бути визначена незалежно від іншої.

6.2.1. Оптимізація хвостової рекурсії

У рекурсії є один великий недолік – кожного разу, коли одна процедура викликає іншу, інформація про виконання викликаючої процедури повинна бути збережена для того, щоб викликаюча процедура могла, після виконання викликаної процедури, відновити виконання на тому ж місці, де зупинилася, що потребує значного об'єму пам'яті.

Слід розглянути спеціальний випадок, коли процедура може викликати себе без збереження інформації про свій стан.

Припустимо, що процедура викликається останній раз, тобто коли викликана процедура завершить роботу, викликаюча процедура не відновить своє виконання. Це значить, що викликаючій процедурі не потрібно зберігати свій стан, тому що ця інформація вже не знадобиться. Як тільки викликана

процедура завершиться, робота процесора повинна йти в напрямі, вказаному для викликаючої процедури після її виконання.

Припустимо, що процедура *A* викликає процедуру *B*, а процедура *B* – процедуру *C* як свій останній крок. Коли *B* викликає *C*, *B* не повинна більше нічого робити. Тому, замість того щоб зберегти в стеку процедури *B* інформацію про поточний стан *C*, можна переписати стару збережену інформацію про стан *B* (яка більше не потрібна) на поточну інформацію про *C*, зробивши відповідні зміни в береженій інформації. Коли *C* завершить виконання, вона вважатиме, що вона викликана безпосередньо процедурою *A*.

Припустимо, що на останньому кроці виконання процедура *B* замість процедури *C* викликає себе. Виходить, що коли *B* викликає *B*, стек (стан) для викликаної *B* повинен бути замінений стеком для викликаючої *B*. Це дуже проста операція, просто аргументам привласнюються нові значення і потім виконання процесу повертається на початок процедури *B*. Тому, з процедурної точки зору, відбувається щось дуже схоже на всього лише оновлення управляючих змінних у циклі.

Ця операція називається *оптимізацією хвостової* рекурсії (tail recursion optimization) або *оптимізацією останнього виклику* (last-call optimization).

6.2.2. Способи реалізації хвостової рекурсії

Фраза «одна процедура викликає іншу, виконуючи свій самий останній крок» на мові Пролог означає, що *виклик є самою останньою підціллю правила або раніше у фразі не було точок повернення*. Правило

```
count(N):-
        write(N), nl,
        NewN = N+l,
        count(NewN).
```

задовольняє обом умовам. Така процедура є хвостовою рекурсією, яка викликає себе без резервування нового стекового фрейма, і тому не виснажує запас пам’яті. Якщо задати цільове твердження

```
count(0).
```

то предикат *count* друкуватиме цілі числа, починаючи з *0*, і ніколи не зупиниться. Кінець кінцем, відбудеться цілочисельне переповнювання, але зупинки через виснаження пам'яті не відбудеться:

```
predicates
count(ulong)
clauses
count(N):-
        write ('\r',N),
        NewN = N+l,
        count(NewN).
goal
nl, count (0).
```

Наступна програма з хвостовою рекурсією печатає список назв академічних груп по рокам для галузі знань «Інформатика та обчислювальна техніка»:

```
predicates
номер(integer,integer,integer)-nondeterm(i,i,i)

clauses
номер(2008,8,5):-!.
номер(N,K,R):-
                write('\r', N, "рік \t \r", K,"-й курс \t група П-", R,"-51\r"),
                NewN=N-1,
                NewK=K+1,
                NewR=R-1,
                NewN>=2007,
                NewK<=5,
                NewR>=7,!,
                номер(NewN,NewK,NewR).
goal
write("Галузь знань ІтаОТ \r_____\r"),
номер(2012,1,12).
```

Рішення Пролог виводить у наступному вигляді:

```
Галузь знань ІтаОТ
_____
2012рік
1-й курс      група П-12-51
2011рік
2-й курс      група П-11-51
2010рік
3-й курс      група П-10-51
2009рік
4-й курс      група П-9-51
2008рік
5-й курс      група П-8-51
```

6.2.3. Причини виникнення не оптимізованої хвостової рекурсії

Існують три помилкові способи організації хвостової рекурсії.

Перший з них – *якщо рекурсивний виклик не є самим останнім кроком, така процедура не є хвостовою рекурсією*. Наприклад:

```
список1(X):-
        write('\r', X),
        NewX=X+l,
        список1 (NewX), nl.
```

Кожного разу, коли предикат *список1* викликає себе, стек повинен бути збережений для того, щоб обробку можна було повернути до викликаючої процедури, яка повинна виконуватися до предиката *nl*. Тому вона зробить всього декілька тисяч рекурсивних викликів до вичерпання пам’яті.

Застосування *Відсікання(!)* не зможе допомогти з *список1*, в якому необхідність створення копій стекових фреймів не пов’язана з неперевіреними альтернативами. Єдиний спосіб удосконалити *список1* – це провести обчислення так, щоб рекурсивний виклик виконувався в кінці речення, тобто прибрати предикат *nl* у кінці правила або перемістити його в інше місце правила.

Другий спосіб зробити хвостову рекурсію не оптимізованою – *залишити деяку можливу альтернативу неперевіреною до моменту виконання рекурсивного виклику*.

Тоді стек повинен бути збережений, оскільки у разі невдалого завершення рекурсивного виклику викликаюча процедура може відкотитися і почати перевіряти цю альтернативу. Наприклад:

```
список2(X):-
        write('\r', X),
        NewX = X+l,
        список2 (NewX).
список2 (X):-
% неперевірена альтернатива
        X < 0,
        write("X - негативне.").
```

Тут перша фраза *список2* викликає себе, коли друга фраза ще не виконана (перевірка коли *X < 0*). Знову програма виснажує пам’ять після певної кількості викликів. В даному випадку вихід один: виконати перевірку у правилі до моменту виконання рекурсивного виклику для оптимізації хвостової рекурсії.

Випадок третій, коли для втрати оптимізації хвостової рекурсії не обов'язково мати неперевірену альтернативу як окрему фразу рекурсивної процедури. *Неперевірена альтернатива може бути і в будь-якому предикаті, що викликається*. Наприклад:

```
список3(X):-
write('\r', X),
NewX = X+1,
умова(NewX),          % неперевірена альтернатива
список3 (NewX).
 % перевірка: NewX уніфікується. з Z
 умова(Z):-Z >= 0,
 умова(Z):-Z<0.
```

Припустимо, що *X* – позитивна величина. Коли *список3* викликає себе, перша фраза *умова* досягає цілі, а друга фраза *умова* ще не перевірена. Тому *список3* повинен зберегти копію свого стекового фрейма, щоб мати нагоду повернутися і почати перевіряти другу фразу *умова* у випадку, якщо рекурсивний виклик завершиться невдало.

Фраза список2 і *список3* гірше, ніж *список1*, тому що вони генерують точки повернення.

Відкидати всі можливі зайві альтернативи дозволяє відсікання *cut*. Таким чином можна виправити фразу *список3:*

```
список3 (X):-
            write('\r', X),
            NewX = X+ 1,
            умова(NewX),
            !,                  % відсікання
            список3 (NewX).
```

Команда «відсікання» означає «одного разу досягнувши цієї крапки, не звертати уваги на альтернативні фрази цього предиката і альтернативні рішення попередніх підцілей в даній фразі». Альтернативи виключаються, фрейм стека не потрібен і рекурсивний виклик може вільно йти далі.

«Відсікання» також ефективне і у фразі *список2*, якщо перемістити перевірку з другої фрази у першу:

```
список2(X) :-
% перевірка у предикаті
X >= 0,
!,                  % відсікання
```

```
write{"\r",X),
NewX=X+1,
cutcount2(NewX).

список2 (X) :-      % тепер не виконується
write("X – негативне.").
```

В початковій версії попереднього прикладу друга фраза повинна була залишати вибір,

```
список2 (X):-
% неперевірена альтернатива
X < 0,
write("X – негативне.").
```

оскільки перша фраза не містила перевірки *X*

```
список2(X):- write('\r', X),
NewX = X+l,
список2 (NewX).
```

Перемістивши перевірку в першу фразу і заперечуючи її, рішення можна ухвалити вже там, а відсікання встановити відповідно до твердження: «Тепер чітко відомо, що не треба писати, що *X* негативний».

Відсікання – це дійсно рішення. Воно використовується всякий раз, коли альтернативи не цікаві.

Те ж стосується *список3*. Предикат *умова* показує ситуацію, коли потрібно вчинити якусь додаткову операцію над *X*, засновану на знаку. Проте код для *умова* недетермінований, і відсікання після його виклику – це вихід. Проте б правильніше, щоб предикат *умова* був детермінований (одне рішення).

Якщо відсікання виконується, комп'ютер припускає, що неперевірених альтернатив немає, і не створює стековий фрейм.

6.2.4. Використовування аргументів як змінних циклу

Для обчислення факторіала мовою С, код записується наступним чином:

```
FactN = 1;
for(int i=1; i<=N; i++) FactN *=i;
```

У мові С символ = є оператором привласнення і означає «привласнити». Фрагмент коду містить 3 змінних:

N – число, факторіал якого обчислюватиметься;

FactN – результат обчислення;

i – лічильник (від 1 до N).

Для більш точного визначення, що відбувається зі змінною *i* на кожному кроці можна замінити цикл за перечисленням *for* на цикл з передумовою *while*:

```
FactN = 1;          % Ініціалізація змінних
i := 1;
While (i <= N)      % Задання циклу: якщо умова ІСТИНА
{
FactN *=i;          % Оновлення FactN та i
i++;
}
```

Реалізація циклу *while* мови *C* у Пролозі виглядає наступним чином:

```
predicates
factorial(unsigned, long)
factorial_aux(unsigned,long,unsigned,long)

clauses
factorial(N, FactN):-
factorial_aux(N,FactN,1,1).
Factorial_aux(N, FactN, I, P) :-
I <= N,
!,                          % відсікання
NewP = P * I,
NewI = 1 + 1,
Factorial_aux(N, FactN, NewI, NewP).
Factorial_aux(N, FactN, I, P):-
I > N,
FactN = P.
```

У фразі для предиката *factorial* є тільки два аргументи — *N* і *FactN*, що вважаються як би вхідним і вихідним значеннями. Фрази для *factorial_aux(N, FactN, I, P)* фактично забезпечують рекурсію. Їх аргументами є чотири змінні, які повинні передаватися з одного кроку в інший. Тому *factorial* просто викликає *factorial_aux*, передаючи йому *N* і *FactN* з початковими значеннями для *i* та *p*:

```
factorial(N, FactN) :-
factorial_aux(N,FactN,1,1).
```

змінні *i* та *p* ініціалізуються.

Але як *factorial* передає змінну *FactN*, у неї поки що немає значення? Відповідь полягає в том, що концептуально Visual Prolog тут уніфікує змінну, названу *FactN* в одній фразі, із змінною, названою *FactN* в іншій фразі. Таким же чином *factorial_aux* передає собі *FactN* як аргумент в рекурсивному виклику.

Кінець кінцем, остання *FactN* отримає значення, і після цього всі інші *FactN*, які уніфікувалися з нею, отримають таке ж значення. Реально є лише одна *FactN*. Visual Prolog може визначити з початкового коду, що *FactN* насправді не використовується перед другою фразою *factorial_aux*, а весь час передається одна і та ж *FactN*.

Предикат *factorial_aux* звичайно перевіряє фразу *«<= N»* для циклічного обчислення, а потім рекурсивно викликає себе з новими значеннями для *i* та *p*. Тут виявляється ще одна особливість Visual Prolog. У Пролозі вірний для арифметики вираз

```
p = p + 1
```

зовсім не є визначенням привласнення (як це повинно бути на більшості імперативних мов програмування).

Неможливо змінити значення змінним в Visual Prolog. Замість цього слід створити нову змінну і надати їй потрібне значення. Наприклад:

```
NewP=P + 1
```

Тому перша фраза виглядає наступним чином:

```
Factorial_aux(N, FactN, I, P) :-
I<=N,
!,
New = P*I,
NewI = I+1,
Factorial_aux(N, FactN, NewI, NewP).
```

У цій фразі відсікання забезпечуватиме оптимізацію хвостової рекурсії, хоча воно і не є останньою фразою в предикаті.

Кінець кінцем *I* буде > *N*, перевищуватиме поточні значення *P*, *FactN* уніфікуються і рекурсія припиниться. Це реалізується у другій фразі, яка виконається, коли перевірка *I <= N* в першій фразі буде неуспішною.

```
Factorial_aux(N,FactN,I,P):- I > N,
FactN = P.
```

Тут немає необхідності робити *FactN=P* окремим кроком, уніфікація може відбуватися в списку аргументів. Підстановка однакової назви змінних вимагає, щоб аргументи в цих позиціях були рівні. Більш того, перевірка *I>N* надмірна, оскільки *I<=N* було перевірено в першій фразі. Це дає завершальну фразу:

```
Factorial_aux(_, FactN _, FactN).
```

Приклад.

```
predicates
шлях(symbol, symbol, integer) - nondeterm (i, o, o), nondeterm (i, o, o),
                                nondeterm (i, i, o)
шлях(symbol, symbol, integer) - nondeterm (i, o, o), nondeterm (i, i, o)
маршрут() - nondeterm ()

clauses
маршрут() :-
шлях("Донецьк", M2, S1),
шлях(M2, M3, S2),
шлях(M3, "Київ", S3),
write ("Донецьк - ",M2, " - ", M3, " - ", "Київ\n"),fail.

маршрут() :-
шлях("Донецьк", M2, S1),
шлях(M2, "Київ", S2),
write ("Донецьк - ",M2, " - ", "Київ\n"), fail.

шлях("Донецьк", "Дніпропетровськ", 180).
шлях("Донецьк", "Ніколаєв", 600).
шлях("Ніколаєв", "Дніпропетровськ", 230).
шлях("Дніпропетровськ", "Київ", 340).
шлях("Ніколаєв", "Київ", 320).


goal
маршрут().

/*
Донецьк - Ніколаєв - Дніпропетровськ - Київ
Донецьк - Дніпропетровськ - Київ
Донецьк - Ніколаєв - Київ
No Solution*/
```

6.3. Рекурсивні структури даних

Рекурсивними можуть бути не тільки фрази, але і структури даних.

Пролог є єдиною популярною мовою програмування, яка дозволяє просто визначити типи рекурсивних даних.

Тип даних є рекурсивним, якщо він допускає структури, що містять такі ж структури, як і вони самі.

Найважливішим рекурсивним типом даних є список, хоча він і не виглядає безпосередньо рекурсивною конструкцією.

Структурою типу даних (рис. 6.1), що вводиться користувачем, є *дерево*.

Важливо, що кожна гілка дерева сама є деревом, тому структура рекурсивна.

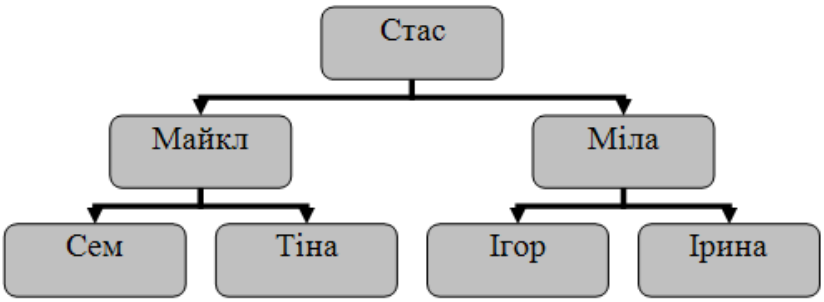


Рисунок 6.1. Дерево – складна структура даних

Рекурсивні типи популяризувалися Ніклаусом Віртом, винахідником мови програмування Pascal. Він не застосовував в Pascal рекурсивні типи, але визначив, якими вони повинні бути. Але якби в Pascal вони все-таки були застосовані, то можна було б визначити дерево на зразок наступного: «Дерево складається з *імені* (*Name*), яке є *рядок* (*string*), а також *лівого* і *правого піддерев*, які теж є деревами». Наступні рядки коду є некоректними в Pascal:

```
tree = record
name: string[80];
left, right: tree
end.
```

Pascal поводиться з деревом як з об’єктом, що складається з вузлів, кожний з яких містить деякі дані і покажчики на два інші вузли.

Visual Prolog дозволяє визначити дійсно рекурсивні типи, в яких покажчики створюються і обробляються автоматично.

Наприклад, можна визначити дерево

```
domains
treetype = tree(string, treetype, treetype)
```

Ця декларація говорить про те, що *дерево записується як функтор tree*, *аргументами якого є рядок і два інші дерева*.

Але це не зовсім задовільна декларація, оскільки немає способу закінчити рекурсію. Насправді дерево не може розповсюджуватися до незкінченності. Деякі вузли не мають зв’язків з подальшими деревами.

У імперативних мовах програмування це можна виразити, привласнивши деяким покажчикам спеціальне нульове значення, але в Пролозі немає доступу до покажчиків. Рішення полягає в тому, щоб визначити два типи дерев – *звичайне* і *порожнє*. Це досягається тим, що *дерево може мати один з двох функторів tree з трьома аргументами або empty без аргументів*.

```
domains
treetype=tree(string,treetype,treetype); empty
(три аргументи)          (немає аргументів)
```

Назви *tree* (функтор, у якого три аргументи) і *empty* (функтор без аргументів) створюються програмістом довільно, і жодному з них немає визначеного у Пролозі значення.

Дерево, представлене на рисунку 6.1, описане наступним чином:

```
tree("Стас",
tree("Майкл",
tree("Сем",empty,empty),tree("Тіна",empty,empty)),
tree("Міла",
tree("Ігор",empty,empty),tree("Ірина",empty,empty))).
```

Для зручності читання програма має підрозділи. Але у Пролозі дерева, при нормальному записі, не вимагається підрозділяти. Точно така ж структура може бути складена іншим способом:

```
tree("Стас",
tree("Майкл",tree("Сем",empty,empty),tree("Тіна",empty,empty)),
tree("Міла",tree("Ігор",empty,empty),tree("Ірина",empty,empty))).
```

Це не є фраза Прологу – це лише складна структура даних.

6.3.1. Обхід дерева

Однією з найбільш часто здійснюваних операцій з деревом є:

- дослідження всіх вузлів і обробка їх деяким чином;
- пошук деякого значення;
- збір всіх значень.

Ці процедури відомі як обхід дерева. Основний алгоритм для цього наступний:

Якщо дерево порожнє, то нічого не робити.

Як і саме дерево, алгоритм є рекурсивним: він обробляє ліве і праве піддерева так само, як і початкове дерево. В Пролозі він виражається двома фразами: одна для порожнього, а інша для непорожнього дерева.

```
обхід(empty).          % нічого не робити

обхід(tree(X,Y,Z)):-
після_визначення_X,
обхід(Y),
обхід(Z).
```

Цей алгоритм спускається по кожній гілці вниз, наскільки можливо, перш ніж повернутися вгору для обходу іншої гілки та відомий як пошук «спочатку – углиб та зліва на право», як показано на рисунку 6.2.

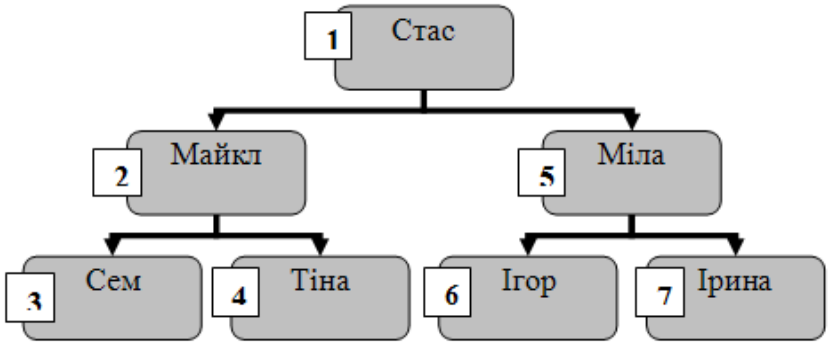


Рисунок 6.2. Алгоритм обходу дерева

Текст програми, у якій реалізований обхід дерева та друк всіх його елементів:

```
domains
типДерево=дерево(string,типДерево,типДерево);порожнє()

predicates
обхід(типДерево)-procedure(i)
clauses
обхід(порожнє).
обхід(дерево(Імя,Ліве,Праве)):-
  write(Імя,'\n'),
  обхід(Ліве),
  обхід(Праве).

goal
обхід(дерево("Стас",
  дерево("Майкл",
  дерево("Сем",порожнє,порожнє),
  дерево("Тіна",порожнє,порожнє)),
  дерево("Міла",
  дерево("Ігор",порожнє,порожнє),
```

дерево("Ірина",порожнє,порожнє)))).

Дерево, показане на рис. 6.2, програма печатає таким чином:

```
Стас
Майкл
Сем
Тіна
Міла
Ігор
Ірина
Yes
```

6.3.2. Створення дерева

Один із способів створення дерева – це вкладена структура з функторів і аргументів. У загальному випадку Пролог створює дерево шляхом обчислення. На кожному кроці *порожнє піддерево* замінюється *непорожнім* в процесі уніфікації.

Створення дерева з одного вузла шляхом привласнення звичайних даних:

створити_дерево(N,tree(N,empty,empty)).

Що означає: «Якщо *N* – дане, то *tree(N,empty,empty)* – це дерево з одного вузла, що містить його».

Побудова структури дерева виконується так само просто. Наступній процедурі потрібні три дерева як аргументи. Вона *вставляє перше дерево як ліве піддерево в друге дерево, і результат цього привласнює третьому дереву*:

створити_ліве(X,tree(A,_,B),tree(A,X,B)).

В цій фразі немає тіла, тобто немає чітких кроків при його виконанні. Все, що необхідно зробити Прологу, – це з'єднати аргументи один з одним в правильному порядку.

Нехай, у наведеному прикладі, необхідно вставити *tree("Майкл",empty,empty)* як ліве піддерево для *tree("Стас",empty,empty)*. Для цього треба виконати цільове твердження:

вставити_ліве(tree("Майкл",empty,empty),tree("Стас",empty,empty),F)

Тоді *F* прийме значення:

tree("Стас",tree("Майкл",empty,empty),empty).

Це є спосіб побудови дерева крок за кроком, що *виконується зліва – направо та знизу – вгору* (з лівого піддерева лівого дерева), наприклад:

```
domains
treetype=tree(string,treetype,treetype);empty()

predicates
створити_дерево(string,treetype) - procedure (i,o)
вставити_ліве(treetype,treetype,treetype) - determ (i,i,o)
вставити_праве(treetype,treetype,treetype) - determ (i,i,o)
виконати - determ ()

clauses
створити_дерево(A,tree(A,empty,empty)).
вставити_ліве(X,tree(A,_,B),tree(A,X,B)).
вставити_праве(X,tree(A,B,_),tree(A,B,X)).

виконати:-
    створити_дерево("Сем",Сем),
    створити_дерево("Тіна",Тіна),
    створити_дерево("Майкл",Майкл),
    створити_дерево("Ігор",Ігор),
    створити_дерево("Ірина",Ірина),
    створити_дерево("Міла",Міла),
    створити_дерево("Стас",Стас),
вставити_ліве(Сем,Майкл,Майкл2),
вставити_праве(Тіна,Майкл2,Майкл3),
вставити_ліве(Ігор,Міла,Міла2),
вставити_праве(Ірина,Міла2,Міла3),
вставити_ліве(Майкл3,Стас,Стас2),
вставити_праве(Міла3,Стас2,Стас3),
            write(Стас3,'\n').

goal
виконати.
```

Результатом є структура:

```
tree("Стас",
 tree("Майкл",tree("Сем",empty,empty),tree("Тіна",empty,empty)),
 tree("Міла",tree("Ігор",empty,empty),tree("Ірина",empty,empty)))
```

6.3.3. Бінарні пошукові дерева

В деревах є можливість зберігати дані так, що їх можна швидко відшукати. Дерево, побудоване для такої цілі, називається *пошуковим* деревом. З погляду користувача, сама структура дерева не несе інформації, дерево – це більш швидка альтернатива списку або масиву.

При обході звичайного дерева розглядається поточний вузол, а потім обидва його піддерева. Щоб знайти певне значення, потрібно розглянути кожний вузол всього дерева.

Час, що затрачується на пошук у звичайному дереві з *N* елементів, в середньому пропорційний *N*.

Бінарне пошукове дерево будується таким чином, що, дивлячись на будь-який вузол, можна передбачити, в якому з його вузлів знаходиться задане значення.

Робиться це завданням відношення порядку між значеннями, таким як алфавітний або пронумерований порядок. *Значення в лівому піддереві передують значенню в поточному вузлі, а в правому — слідують після значення в поточному вузлі* (рис. 6.3).

Ті ж числа, встановлені в іншому порядку, дадуть інше дерево.

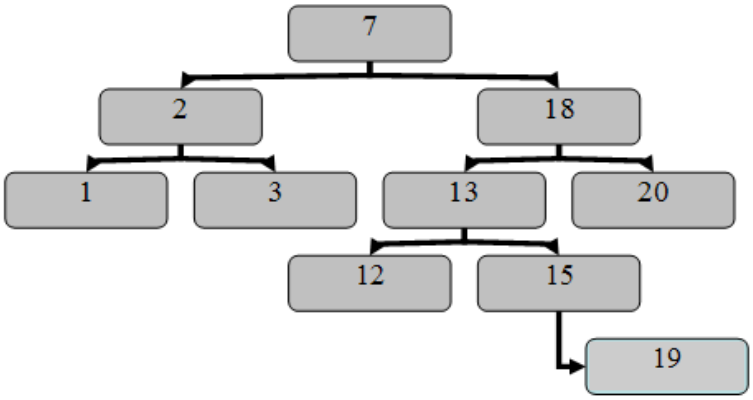


Рисунок 6.3. Бінарне пошукове дерево

Можна виключити з розгляду половину вузлів, що залишилися, і провести пошук дуже швидко. Якщо тепер розмір дерева збільшити удвічі, то для пошуку буде потрібно тільки один додатковий крок.

Час, що вимагається для пошуку значення у бінарному пошуковому дереві, в середньому пропорційний $\log_2 N$ (а насправді пропорційно logN по будь-якій основі).

Щоб побудувати дерево, потрібно почати з порожнього дерева і додавати до нього значення одне за іншим. Процедура додавання значення така ж, як при пошуку: необхідно просто знайти місце, де це значення повинне знаходитися, і вставити його туди. Цей алгоритм полягає в наступному:

Якщо поточний вузол є порожнє дерево, то вставити в нього значення.

Інакше, порівняти значення, яке необхідно вставити, із значенням в поточному вузлі. Вставити значення в ліве або праве піддерево, залежно від результату порівняння.

У Пролозі цьому алгоритму потрібно три фрази – по одній на кожний випадок. Перша фраза така

```
вставити(NewItem,empty,tree(NewItem,empty,empty):-!.
```

що природною мовою означає:

«Результатом вставки *NewItem* (нового значення) в *empty* (порожнє дерево) буде Дерево

```
tree(NewItem,empty,empty)».
```

! – відсікання означає, що якщо цю фразу можна успішно застосувати, то інші фрази перевіряти не треба.

Наступні друга і третя фрази здійснюють вставку в непорожні дерева:

```
вставити (NewItem,tree(Element,Left,Right),
          tree (Element,NewLeft,Right):-
          NewItem<Element,!,
          вставити(NewItem,Left,NewLeft).

вставити (NewItem,tree(Element,Left,Right),
          tree (Element,Left,NewRight):-
          вставити(NewItem Right,NewRight).
```

Якщо *Newitem<Element*, то його потрібно вставляти у ліве піддерево, а якщо інакше *Newitem>Element*, то у праве піддерево. Багато роботи виконується перевіркою відповідності аргументів у голові правила.

6.3.4. Сортування на основі дерева

Після того, як дерево побудовано, можна легко переставити всі його елементи в алфавітному порядку. Алгоритм для цього – варіант пошуку «спочатку — углиб»:

Якщо дерево порожнє, то нічого не робити.

Інакше, переставити всі елементи лівого піддерева, потім поточний елемент, потім всі елементи правого піддерева.

Мовою Пролог:

```
сортувати(empty).     % Нічого не робити
сортувати(tree (Item,Left,Right)):-
сортувати(Left),
поточний_елемент(Item),
сортувати(Right).
```

Сортування наступних один за одним з значень можна виконати, вставляючи їх в дерево, і потім переставляючи їх по порядку. Для N значень це займе час, пропорційний N log N, оскільки вставка і перестановка займає час, пропорційний logN, причому те і інше повинне бути виконане за N кроків. Це найшвидший сортуючий алгоритм, відомий на сьогодні.

Запитання. Завдання

1. У який спосіб мовою Visual Prolog реалізується повтор?
2. Яка відмінність між відкатом і рекурсією?
3. Яким чином можна усунути проблему використання рекурсією додаткового об'єму пом'яті?
4. У яких випадках у пролог-програмах застосовують предикат *repeat*?
5. Пояснити термін «*Рекурсивна процедура*».
6. Пояснити логіку рекурсії.
7. З чого складається рекурсивна процедура?
8. Як уникнути зациклення при виконанні рекурсії?
9. Як пояснити рекурсивну операцію «оптимізація останнього виклику»?
10. Як отримати «хвостову рекурсію»?
11. Як змінити значення змінним в Visual Prolog?
12. Який тип даних є рекурсивним?
13. Декларація структури типу «дерево».
14. Які операції можна здійснювати з деревом?
15. Використання бінарних пошукових дерев?

Тест для самоконтролю знань з розділу 6[F]

1. Крок рекурсії – це ...?
   а) фраза, що визначає якусь початкову ситуацію або ситуацію у момент припинення;

[F] Усі запитання мають один правильний варіант відповіді.

б) найменше відношення, що володіє властивістю транзитивності;
в) правило, в тілі якого обов'язково міститься, як подцілі, виклик предиката, що визначає транзитивність;
г) виконання або невиконання будь-якої умови.

2. Транзитивне замикання – це ...?
   а) фраза, що визначає якусь початкову ситуацію;
   б) найменше відношення, що володіє властивістю транзитивності;
   в) правило, в тілі якого обов'язково міститься виклик предиката, що визначає транзитивність;
   г) виконання будь-якої умови.
3. Використання в рекурсивних процедурах Пролог аргументів як змінних циклу. Як оновлюються циклічні змінні при використанні ітерацій?
   а) оновлення циклічних змінних не відрізняється від обчислення в імперативних мовах. Наприклад P=P*I;
   б) не можливо змінити змінну в Пролог, слід створити нову. Наприклад NewP=P*I;
   в) змінити змінну в Пролог, створивши нову. Наприклад P=P*I;
   г) спосіб відсутній.
4. Метод виконання пошуку з поверненням для виконання ітерацій?
   а) предикати repeat. repeat:-repeat.
   б) предикат typewriter( ).
   в) оператор For.
   г) предикат readchar( ).
5. Рекурсія дозволяє використовувати в процесі визначення предиката його самого. Це означає що ...
   а) комп'ютер копіює і зберігає в стеку код виконання, всі аргументи і проміжні значення;
   б) комп'ютер створює нову копію предиката таким чином, що він здатний викликати сам себе як самостійну процедуру;
   в) комп'ютер копіює і зберігає в стековому фреймі код виконання рекурсивного кроку;
   г) комп'ютер копіює і зберігає постійно в стековому фреймі всі аргументи і проміжні значення.
6. Як задати хвостову рекурсію ?
   а) у рекурсивній процедурі не допускається пошук з поверненням, отже, не повинно бути точок відкату;
   б) необхідно, щоб рекурсивний виклик був останньою подціллю фрази;
   в) рекурсивний виклик завжди повинен бути передостанньою подціллю фрази, якщо раніше в фразі не було точок відкату;
   г) рекурсивний виклик повинен бути останньою подціллю фрази, якщо раніше у фразі не було точок відкату.
7. Рекурсивна процедура – це ...?
   а) процедура, яка викликає сама себе;

б) процедура, що містить в своїй структурі складені об'єкти данних;
в) процедура, що містить в своїй структурі складені об'єкти даних, що оголошені більш ніж одним типом;
г) процедура, що містить в своїй структурі складені об'єкти даних, що оголошені одним типом даних.

8. Базис рекурсії – це ...?
   а) фраза, що визначає якусь початкову ситуацію або ситуацію у момент припинення;
   б) найменше відношення, що володіє властивістю транзитивності;
   в) правило, в тілі якого обов'язково міститься, як подцілі, виклик предиката, що визначає базис;
   г) виконання або невиконання якоїсь умови.
9. Чим відрізняється хвостова рекурсія від звичайної рекурсивної процедури?
   а) хвостова рекурсія викликає себе із збереженням параметрів кожного виклику в стековому фреймі і виснажує запас пам’яті;
   б) відмінність тільки в записі предикатів правила рекурсивної процедури;
   в) хвостова рекурсія викликає себе без резервування нового стекового фрейма і не виснажує запас пам’яті;
   г) відмінностей немає.
10. Як будується бінарне пошукове дерево?
   а) значення у правому піддереві передують значенню в поточному вузлі, а у лівому – слідують після значення в поточному вузлі;
   б) значення в лівому піддереві передують значенню в поточному вузлі, а в правому – слідують після значення в поточному вузлі;
   в) значення у правому і лівому піддереві передуют значенню в поточному вузлі;
   г) правил розміщення значень у вузлах не існує.
11. Чому ріний час, що затрачується на пошук у звичайному дереві, що складається з N елементів?
   а) в середньому пропорційний Iog_2N;
   б) пропорційний logN при будь-якій основі;
   в) в середньому пропорційний N.
12. Алгоритм, за яким виконується обхід звичайного дерева?
   а) значення в лівому піддереві передують значенню в поточному вузлі, а в правому – слідують після значення в поточному вузлі;
   б) значення у правому піддереві передують значенню в поточному вузлі, а у лівому – слідують після значення в поточному вузлі;
   в) спускається по кожній гілці вниз, наскільки можливо, перш ніж повернутися вгору для обходу іншої гілки;
   г) спускається по кожній гілці вниз на рівні вузла, перш ніж повернутися вгору для обходу іншої гілки.

РОЗДІЛ 7. СПИСКИ

В імперативних мовах, як правило, основною структурою даних є масиви. В Пролозі так само, як і в Ліспі, основним складовим типом даних є список.

Неформальне визначення списку – впорядкована послідовність елементів довільної довжини.

Списки можна грубо порівняти з масивами в інших мовах програмування, але, на відміну від масивів, для списків немає необхідності наперед оголошувати їх розмір. Звичайно, є й інші способи об'єднати декілька об'єктів в один:

- якщо число об'єктів наперед відоме, то можна зробити їх аргументами однієї складової структури даних;
- якщо число об'єктів не визначено, то можна використовувати рекурсивну складову структуру даних, таку як дерево.

Але звичайно працювати із списками легше, оскільки Visual Prolog забезпечує для них більш чіткий запис.

Список задається переліком *елементів* списку через кому в квадратних дужках, як наведено нижче у прикладах.

[monday, tuesday, wednesday, thursday, friday, saturday, sunday] – список, елементами якого є англійські назви днів тижня;

["понеділок", "вівторок", "середа", "четвер", "п'ятниця", "субота", "неділя"] – список, елементами якого є українські назви днів тижня;

[1, 2, 3, 4, 5, 6, 7] – список, елементами якого є номери днів тижня;

['п', 'в', 'с', 'ч', 'п', 'с', 'н'] – список, елементами якого є перші символи назв днів тижня українською;

[ ] – порожній список, тобто список, що не містить елементів.

Елементи списку можуть бути будь-якими, у тому числі і складовими об'єктами. Зокрема, *елементи списку самі можуть бути списками*.

У розділі опису доменів списки описуються таким чином:

domains

*<ім'я спискового домена>=< ім'я домена елементів списку>**

У слова «список» немає спеціального значення у Visual Prolog. Зірочка після імені домена указує на те, що описується список, який складається з об'єктів відповідного типу. Наприклад:

```
listI=integer* /* список, елементи якого –  цілі числа */
listR=real* /* список, що складається з речовинних чисел */
listC=char* /* список символів */
lists=string* /* список, що складається з рядків */
listL=listI* /* список, елементами якого є  списки цілих чисел */
```

Останньому прикладу відповідатимуть списки вигляду:

```
[[1,3,7],[ ],[5,2,94][–5,13]]
```

В класичному Пролозі елементи списку можуть належати різним доменам, наприклад: *[monday, 1, "понеділок"]*

У Visual Prolog, у зв'язку із строгою типізацією, всі елементи списку повинні належати одному домену. Декларація домена для елементів, що належать єдиному типу або являються набором відмінних один від одного елементів, відмічених різними функторами, повинна відповідати формату:

```
domains
elementlist=elements*
elements= .... (тип)
```

У Visual Prolog не можна змішувати стандартні типи у списку. Наприклад, наступна декларація неправильно визначає список, складений з елементів, що є цілими і дійсними числами або ідентифікаторами:

```
elementlist=elements*
elements=integer;real;symbol /* Невірно */
```

Проте можна розмістити в одному списку об'єкти різних типів, використовуючи домен з відповідними альтернативами. Щоб оголосити список, складений з цілих, дійсних і ідентифікаторів, треба визначити один тип, який включає всі три типи з *функторами*, які покажуть, до якого типу відноситься той або інший елемент. Наприклад, наступне описання:

```
domains
element=i(integer);c(char);s(string) % функторами є: i, c та s
listE=element*
```

дозволить мати справу із списками вигляду

```
[i(–15),s("Студент"),c('з'),s("групи П-09-51"),c('+'),s("КР"),i(90),c('!')]
```

Список являється рекурсивним складеним об'ктом.

Рекурсивне визначення списку – це структура даних, що визначається таким чином:

- порожній список [ ] є списком;
- структура вигляду [H|T] є списком, якщо *H* – перший елемент списку (або декілька перших елементів списку, перерахованих через кому), а *T* – список, що складається з елементів початкового списку, що залишилися.

Прийнято називати *H головою списку*, а *T – хвостом списку*. Помітимо, що вибір змінних для позначення голови і хвоста не випадковий (англійською голова – *Head*, а хвіст – *Tail)*.

Фактично операція «|» (використовується замість коми) дозволяє розділити список на *хвіст* і *голову* або, навпаки, приписати об'єкт (об'єкти) до початку списку. Можна використовувати обидва роздільники в одному і тому ж списку за умови, що символ вертикальної риски буде останнім з роздільників.

Хвіст списку – завжди список, голова списку – завжди елемент.

Дане визначення дозволяє організовувати рекурсивну обробку списків, розділяючи непорожній список на голову і хвіст. Хвіст, у свою чергу, також є списком, що містить меншу кількість елементів, ніж початковий список. Якщо хвіст не порожній, його також можна розбити на голову і хвіст. І так до тих пір, поки не дістанемося до порожнього списку, у якого немає голови.

Наприклад, в списку [1,2,3] елемент 1 є головою, а список [2,3] – хвостом, тобто [1,2,3] = [1|[2,3]].

Помітимо, що хвіст цього списку [2,3], у свою чергу, може бути представлений у вигляді голови 2 і хвоста [3], а список [3] можна розглядати у вигляді голови 3 і хвоста [ ]. Порожній список далі не розділяється.

У результаті одержуємо, що список [1,2,3] еквівалентний списку [1|[2,3]], який, у свою чергу, еквівалентний списку [1|[2|[3]]].

Останній зіставимо із списком [1|[2|[3|[ ]]]].

В цьому ж списку можна виділити два перші елементи і хвіст з третього елемента [1,2|[3]]. І, нарешті, можливий варіант розбиття на голову з трьох перших елементів і порожній хвіст: [1,2,3|[ ]].

В одноелементному списку [a] голова є a , а хвіст – [ ].

Список [a,b,c] еквівалентний спискам:

```
[a| [b, c]]
[a| [b| [c] ] ]
[a| [b| [c| [] ] ] ]
```

Якщо, наприклад, зі списку [a,b,c] вибирати перший елемент списку достатню кількість разів, далі слідує порожній список [ ], який не можна розділити на голову і хвіст. Це значить, що список має структуру дерева, як і інші складові об'єкти. Структура одноелементного дерева та дерева [a,b,c] представлена на рис. 7.1

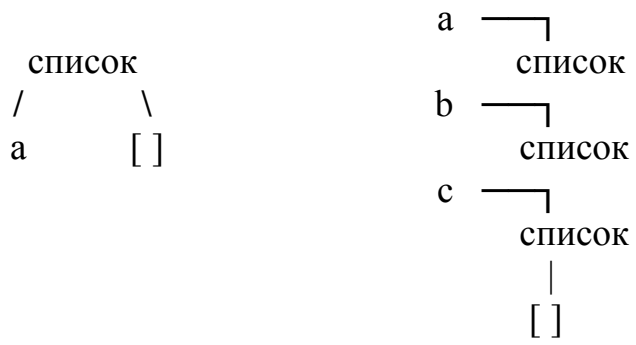


Рисунок 7.1. Структура дерева

У списках змінним присвоюються значення, наприклад:

Список1	Список2	Присвоювання
[X,Y,Z]	[іван,читає,книгу]	X=іван, Y=читає, Z=книгу
[7]	[X|Y]	X=7, Y=[]
[1,2,3,4]	[X,Y,|Z]	X=1, Y=2, Z=[3,4]

7.1. Використання списків

Список є рекурсивною складовою структурою даних, тому потрібні алгоритми для його обробки. Головний спосіб обробки списку – це перегляд і обробка кожного його елемента, поки не буде досягнутий кінець списку.

Алгоритму цього типу звичайно потрібно дві фрази. Перша говорить, що робити із звичайним списком (списком, який можна розділити на голову і хвіст), друга – що робити з порожнім списком.

7.1.1. Друк списків

Друк списків (наприклад числового типу) виконується за правилами, як наведено у наступному тексті Пролог-програми:

```
Domains
 список=integer*                % або дані будь-якого домену

predicates
 друкувати_список (список)      % оголошення домена користувача

clauses
 друкувати_список([ ]).         % Якщо список порожній, то нічого не робити
 друкувати_список ([H|T]):-     % Привласнити змінну  H – голові, T – хвосту
 write (H) ,nl,                 % Друкувати голову списку
 друкувати_список (T).          % Друкувати хвіст списку

goal
 друкувати_список ([1,2,3]).
```

Цільове твердження друкувати_список ([1,2,3]) задовольняє друге правило при H=1 и T=[2,3]. Пролог друкує 1 і визве рекурсивно друкувати_список(T), виклик якого (друкувати_список ([2,3])) знову задовольняє друге правило, де H=2 та T=[3] – виконується друк 2 та знову рекурсивно визивається друкувати_список. Цього разу виклик з цільовим твердженням друкувати_список([3]), голова якого 3, а хвіст [ ] – Пролог печатає 3 та виконує рекурсивний виклик друкувати_список([ ]), який відповідає першому правилу (порожній список не можна розділити на голову і хвіст) та виконується без подальших дій.

7.1.2. Довжина списку

Довжина списку – це кількість його елементів, що визначається як 1 плюс довжина його хвоста. Наступний текст програми демонструє підрахунок елементів списку [1,2,3], де використовується не хвостова рекурсія, так як рекурсивний виклик не є останнім кроком у фразі:

```
domains
 список=integer*

predicates
```

```
 довжина(список,integer)

clauses
 довжина([ ],0).        % якщо список порожній - кількість елементів списку = 0
 %Значення голови списку не важливо – це є 1 елемент
 довжина ([_|T], L):- % – хвіст, L – кількість елементів списку
 довжина (T, Довжина_хвоста),
 L=Довжина_хвоста + 1.

goal
 довжина([1,2,3],L).
```

Цільове твердження відповідає другому правилу при T=[2, 3] для підрахунку довжини хвоста T, тоді змінна Довжина_хвоста набуде значення 2, а змінна L – значення 3.

Процедура довжина визиває сама себе рекурсивно. Проміжний крок виконується з цільовим твердженням довжина([2,3],Довжина_хвоста) у другому правилі, де [3] привласнюється T, а Довжина_хвоста привласнюється L. Змінні Довжина_хвоста у цільовому твердженні і у правилі не співпадають, тому що кожен рекурсивний виклик у правилі має власний набір змінних.

Цільове твердження полягає у визначенні довжини списку [3], таким чином довжина викликає сама себе рекурсивно, щоб отримати довжину списку [3]. Хвіст [3] порожній [ ], тому T привласнюється [ ]. З цільового твердження довжина ([ ], Довжина_хвоста) необхідно визначити довжину [ ] та додати до неї 1 і отримати [3], що задовольняє першому правилу, привласнить нульове значення змінній Довжина_хвоста. Visual Prolog додасть до нуля 1 та отримає довжину списку [3] і повернеться до визиваючого правила, яке знову додасть 1 та отримає довжину списку [2,3]. Аналогічно, після повернення до визиваючого правила, додасть 1 та отримає довжину списку [1,2,3].

Змінні з одним ідентифікатором або з різних викликів одного правила відрізняються одна від одної:

```
довжина([1,2,3],L1).
довжина([2,3],L2).
довжина([3],L3).
довжина([ ],0).
L3=0+1=1
L2=L3+1=2
LI=L2+1=3
```

Предикат для визначення довжини списку можна виразити через хвостову рекурсію. Даний спосіб передбачає використання предиката, аргументами якого повинні бути:

- список, який з кожним викликом зменшуватиметься на один елемент, доки він не стане порожнім;
- вільний параметр, який зберігатиме проміжні результати (довжину списку);
- лічильник, який розпочинаючись з нульового значення буде зростати на одиничку при кожному рекурсивному виклику.

Коли список стане порожнім, лічильник уніфікується з вільним параметром, як наведено у тексті наступної програми:

```
domains
 список=integer*

predicates
 довжина(список,integer,integer)

clauses
 довжина([ ],Проміжний_результат,Проміжний_результат).
 довжина([_|Т],Проміжний_результат,Лічильник):-
      НовеЗначенняЛічильника=Лічильник+1,
      довжина(Т,Проміжний_результат,НовеЗначенняЛічильника).

goal
 довжина([1,2,3],L,0).   %  початкове значення лічильника =0
```

7.1.3. Перетворення списку

Будь-який список можна змінити, застосовуючи якусь дію для елементів цього списку, наприклад отримати новий список з елементами, які стануть більшими на 10 одиниць:

```
domains
 список=integer*

predicates
 заміна(список,список)

clauses
 %умова, за якої нічого не виконувати
 заміна([ ],[ ]).
 % відокремити голову списку
 заміна ([Голова|Хвіст],[ГоловаНовий|ХвістНовий]):-
```

```
  %додати 10 до елемента 1-го списку
  Число=10,
  ГоловаНовий=Голова+Число,
  %викликати елемент із залишку списку
  заміна(Хвіст,ХвістНовий).

goal
  заміна([1,2,3,4],НовийСписок).
```

Для того, щоб додати *Число* до всіх елементів порожнього списку, потрібно створити новий порожній список.

Щоб додати *Число* до всіх елементів будь-якого непорожнього списку, треба додати *Число* до голови і зробити отриманий елемент головою результуючого списку, потім додати *Число* до кожного елемента хвоста списку і зробити це хвостом результату.

Процедура *заміна* є хвостовою рекурсією, тому що по завершенню виконання *ГоловаНовий* та *ХвістНовий* вже є головою і хвостом результату, і немає окремої операції для їх об'єднання. Таким чином, рекурсивний виклик є останнім кроком.

7.1.4. Перетворення списку з застосуванням критеріїв

У Visual Prolog існує можливість отримати новий список з існуючого, не виконуючи однакових дій з кожним елементом списку, якщо до останнього застосувати умову. Наприклад, з числового списку отримати новий, у якому значення елементів будуть позитивними.

```
domains
  список=integer*

predicates
  новийСписок(список,список)

clauses
  новийСписок([ ],[ ]).

  новийСписок ([H|T],ПеревіркаХвоста):-
  H < 0,                        % Перевірка
  !,                            % Нічого не виконувати
  новийСписок(T,ПеревіркаХвоста).
  новийСписок([H|T],[H|ПеревіркаХвоста]):-
  новийСписок(T,ПеревіркаХвоста).

goal
```

```
новийСписок([2,30,-3,10,-45,36, 3, 0,56,48,68],X).
```

Результатом виконання програми є новий список:

```
X=[2,30,10,36,3,0,56,48,68]
1 Solution
```

7.1.5. Читання списків. Приналежність до списку

Для того щоб перевірити, чи є заданий елемент у списку, треба перевірити відношення між двома аргументами – елементом і списком елементів, яке можна виразити предикатом:

```
належить(назва,список).
```

де *назва* аргумент належить списку *список*.

Алгоритм реалізації складається з двох фраз.

Перша з них перевіряє відповідність *Назва* голові списку, і у разі збігу у даному випадку немає необхідності перевіряти хвіст списку. Інакше використовується друга фраза, коли голова списку не відповідає *Назва*, тоді слід перевірити хвіст списку:

```
domains
 список=назва *
 назва=symbol

predicates
 приналежність(назва,список)

clauses
 приналежність(Назва,[Назва|_]).

 приналежність(Назва,[_|Tail]):-
 приналежність(Назва,Tail).

goal
 приналежність (“Жовті Води”,[“Київ”, “Дніпропетровськ”, “Жовті Води”,
                                                    “Харків”]).
```

Якщо дане цільове твердження пропонує Прологу перевірити, чи вірне твердження, що вказаний елемент “*Жовті Води*” відноситься до списку, то наступний запис цільового твердження

```
приналежність(X,["Київ","Дніпропетровськ","Жовті Води","Харків"]).
```

запропонує знайти всі елементи списку.

7.1.6. Об'єднання списків

Для об'єднання двох списків необхідно створити предикат, який об'єднує три аргументи

```
приєднати(Список1,Список2,Список3)
```

У випадку, коли *Список1* виявиться порожнім, результатом об'єднання залишиться *Список2*. Мовою Пролог запис має вигляд

```
приєднати([ ],Список2,Список2).
```

Якщо *Список1* не порожній, то можна об'єднати *Список1* і *Список2* для формування *Список3*, зробивши *Список1* головою списку Список3. У наступному твердженні змінна H використовується як голова для *Список1* і для *Список3*. Хвіст *Список3* – це *L3*, він складається з об'єднання залишку *Список1* (тобто *L1*) і всього *Список2*:

```
приєднати([H|L1],Список2[H|L3]):-приєднати(LI,Список2,L3).
```

Предикат приєднати виконується таким чином: поки *Список1* не порожній, рекурсивна фрага передає по одному елементу в *Список3*. Коли *Список1* стане порожнім, перша фраза уніфікує *Список2* з отриманим *Список3*:

```
domains
 integerlist=integer*

predicates
 приєднати(integerlist,integerlist,integerlist)

clauses
 приєднати([ ],Список,Список).
 приєднати([HIL1],Список2,[H|L3]):-приєднати (L1,Список2,L3).

goal
 приєднати([1,2,3,4,5], [6,7,8,9],L).
```

Результат роботи Пролог:

```
L=[1,2,3,4,5,6,7,8,9]
1 Solution
```

Для наступного цільового твердження

```
приєднати([1,2],[3],L),приєднати(L,L,LL).
```

пролог надрукує результат:

```
L=[1,2,3],LL=[1,2,3,1,2,3]
1 Solution
```

Предикатом приєднати визначене відношення між трьома списками, яким виконується об'єднання у третьому списку перших двох. Але це відношення зберігається і в інших випадках, як би були відомі перший і третій список, а другий невідомий.

7.1.7. Поділ списків

Вищенаведені правила будуть справедливі і у випадку, коли відомий результуючий список. Для визначення, які з двох списків необхідно об'єднати для отримання відомого списку, треба написати наступний текст програми:

```
domains
 integerlist= nteger*

predicates
 поділити(integerlist,integerlist,integerlist)

clauses
 поділити([],Список,Список).
 поділити([H|L1],Список2,[H|L3]):-поділити(L1,Список2,L3).

goal
 поділити(LI,L2,[1,2,3,4]).
```

Пролог надрукує всі можливі варіанти списків:

```
LI=[],L2=[1,2,3,4]
LI=[1],L2=[2,3,4]
LI=[1,2],L2=[3,4]
LI=[1,2,3],L2=[4]
LI=[1,2,3,4],L2=[ ]
5 Solutions
```

У предиката *поділити* є можливість працювати з різними потоками параметрів, залежно від того, які початкові дані йому задають. Проте не для всіх предикатів є можливість бути викликаними з різними потоками параметрів. Якщо фраза Прологу може бути використана з різними потоками параметрів, вона називається *оборотною* фразою, яка має додаткові переваги і додає «потужності» предикатам.

Як приклад, можна знайти один із списків, які були об'єднані для отримання третього списку:

```
goal
поділити(LI,[3,4], [1,2,3,4]).
```

Пролог знайде одне рішення:

```
LI=[1,2]    1 Solution
```

7.1.8. Сортування списків

Один із способів сортування – бульбашкою, ідея якого полягає в тому, що, на кожному кроці порівнюються два сусідні елементи списку. Якщо виявляється, що вони стоять неправильно, тобто попередній елемент менше наступного, то вони міняються місцями. Цей процес продовжується до тих пір, поки є пари сусідніх елементів, розташовані в неправильному порядку. Це і означатиме, що список відсортований – при кожному проході мінімальні елементи переміщуються до початку списку.

Сортування бульбашкою реалізується за допомогою двох предикатів. Один з них, назвемо його *перестановка*, порівнює два перші елементи списку і у випадку, якщо перший виявиться більше другого, міняє їх місцями. Якщо ж перша пара розташована в правильному порядку, цей предикат переходитиме до розгляду хвоста.

Основний предикат *визиваючий_предикат* здійснюватиме бульбашкове сортування списку, використовуючи допоміжний предикат *перестановка*, як наведено у наступній пролог-програмі:

```
domains
 list=integer*

predicates
 перестановка(list,list)
 визиваючий_предикат(list,list)

clauses
 перестановка([X,Y|T],[Y,X|T]):-
 X>Y,!.
          /* переставляємо перші два елемента,
          якщо перший більший за другий */
 перестановка([X|T],[X|T1]):-перестановка(T,T1).
          /*переходимо до перестановок у хвості*/
 визиваючий_предикат(L,L1):-перестановка(L,LL),
                            /* визиваємо предикат, який
                            здійснює перестановку */
 !,
 визиваючий_предикат(LL,L1). /* спробуємо ще раз відсортувати
                                отриманий список */
 визиваючий_предикат(L,L).
/* якщо перестановок не було, це означає список відсортований */
```

```
goal
 визиваючий_предикат([5,8,3,4,1,9],L).
```

Метод працює доки є хоча б одна пара елементів списку, розміщена в неправильному порядку. Як тільки такі елементи закінчилися, предикат *перестановка* терпить невдачу, а *визиваючий_предикат* переходить від правила до факту і повертає як другий аргумент відсортований список:

```
L=[1,3,4,5,8,9]
1 Solution
```

Інший спосіб – сортування вставкою. Він заснований на тому, що якщо хвіст списку вже відсортований, то достатньо поставити перший елемент списку на його місце в хвості, і весь список буде відсортований. При реалізації цієї ідеї необхідно створити два предикати.

Задача першого предиката *вставка* – вставити значення (голову початкового списку) у вже відсортований список (хвіст початкового списку), так щоб він залишився впорядкованим. Його першим аргументом буде значення, що вставляється, другим – відсортований список, третім – список, отриманий вставкою першого аргументу в потрібне місце другого аргументу так, щоб не порушити порядок.

Предикат *вст_сорт*, власне, і організовуватиме сортування початкового списку методом вставок. Як перший аргумент йому дають довільний список, який потрібно відсортувати. Другим аргументом він повертає список, що складається з елементів початкового списку, що стоять в правильному порядку:

```
domains
 list=integer*

predicates
 вст_сорт(list,list)
 вставка(integer,list,list)

clauses
 вст_сорт([ ],[ ]). /* відсортований порожній список залишається порожнім*/
 вст_сорт([H|T],L):-
 вст_сорт(T,T_Sort),
        /* T - хвіст вхідного списку, T_Sort - відсортований хвіст
        вхідного списку */
 вставка(H,T_Sort,L).
        /* вставляємо H (перший елемент вхідного списку) в T_Sort,
        отримуємо L (список, що складається з елементів вхідного
```

```
                    списку, розміщених за зменшенням ) */
вставка(X,[ ],[X]). /* при вставці будь-якого значення в порожній
                    список, отримуємо одноелементний список */

вставка(X,[H|T],[H|T1]):-
X>H,!,      /* якщо значення, яке вставляється, більше голови списку,
               це означає що його потрібно вставляти у хвіст */
вставка(X,T,T1). % вставляємо X у хвіст T, в результаті отримуємо список T1

вставка(X,T,[X|T]).
/* ця фраза (за рахунок відсікання у попередньому правилі)
    виконується, тільки якщо значення, яке вставляється,
   не більше голови списку T, значить, добавляємо його
   першим елементом  у список T */

goal
 вставка(25,[5,9,14,22,29],L).
```

Результатом є одне рішення – новий список:

```
L=[5,9,14,22,25,29]
1 Solution
```

Наступний метод сортування списків – швидке – сортування з досить високою ефективністю.

Вибирається деякий критичний (з існуючого списку) елемент, щодо якого розбивається початковий список на два підсписки. В один поміщаються елементи, менші критичного елемента, в другій – більші або рівні. Кожний з цих списків сортується тим же способом, після чого приписується до списку тих елементів, які менше критичного, спочатку сам критичний елемент, а потім – список елементів більших за критичний. Результатом є список, що складається з елементів, які розміщуються в правильному порядку.

Метод виражається через два предикати.

Допоміжний предикат *перестановка* відповідатиме за розбиття списку на два підсписки. У нього будуть чотири аргументи. Перші два елементи – вхідні: перший – початковий список, другий – критичний елемент. Третій і четвертий елементи – вихідні, відповідно, список елементів початкового списку, які менше критичного, і список, що складається з елементів, які більше критичного елемента.

Предикат *швидке_сорт* реалізовує алгоритм швидкого сортування і складається з двох фраз. Правило здійснюватиме за допомогою предиката

перестановка розділення непорожнього списку на два підсписки, потім сортує кожний з цих підсписків рекурсивним викликом себе самого, після чого, використовуючи предикат *консолідація*, конкретизує другий аргумент списком, одержуваним об'єднанням відсортованого першого підсписку і списку, сконструйованого з критичного елемента (голови початкового списку) і відсортованого другого підсписку, як наведено у наступному тексті програми:

```
domains
  list=integer*

predicates
  швидке_сорт(list,list)
  перестановка(list,integer,list,list)
  консолідація(list,list,list)

clauses
  швидке_сорт([ ],[ ]). /* відсортований порожній список
                           залишається порожнім списком */

  швидке_сорт([H|T],O):-
  перестановка(T,H,L,G),    /* розділити список T на L (список елементів
                               менших критичного елемента H) і G (список
                               елементів більших H) */
  швидке_сорт(L,L_s),    /* список L_s - результат впорядкування елементів
                               списку L  */
  швидке_сорт(G,G_s),  /* аналогічно, список G_s - результат впорядкування
                               елементів списку G  */
  консолідація(L_s,[H|G_s],All). /* приєднати список L_s до списку, у якого
                                    голова H, а хвіст G_s, результат позначити
                                    через All */
  консолідація([ ],[ ],[ ]).       /* консолідація порожніх списків*/
  перестановка([ ],_,[ ],[ ]).   /* при поділу порожніх списків
                               результатом будуть порожні списки */
  перестановка([X|T],Y,[X|T1],Bs):-
                  X<Y,!,
                  перестановка(T,Y,T1,Bs).
                        /* якщо елемент X менше критичного елемента Y,
                        то слід додати його в третій аргумент */
  перестановка([X|T],Y,T1,[X|Bs]):-
                  перестановка(T,Y,T1,Bs).
                        /* інакше додати його в четвертий аргумент */
goal
  перестановка([5,25,10,15,30,20],25,L_s,G_s).
```

Результатом роботи Пролог-програми є списки:

```
L_s=[5,10,15,20], G_s=[25,30]
1 Solution
```

Найдавніший з алгоритмів сортування – метод злиття. Ідея цього методу полягає в поділі списку, який потрібно упорядкувати, на два підсписки, упорядкування кожного з них цим же методом, після чого злиття впорядкованих підсписків назад в один загальний список.

Спершу необхідно створити предикат, який ділитиме початковий список на два. Він складатиметься з двох фактів і правила. Перший факт затверджуватиме, що порожній список можна розбити тільки на два порожні підсписки. Другий факт пропонуватиме розбиття одноелементного списку на той же одноелементний список і порожній список. Правило працюватиме у випадках, не охоплених фактами, тобто коли упорядковуваний список містить не менше чим два елементи. В цій ситуації слід відправляти перший елемент списку в перший підсписок, другий елемент – в другий підсписок, і продовжувати розподіляти елементи хвоста початкового списку.

Основний предикат, який, власне, і здійснюватиме сортування списку, складатиметься з трьох фраз. Перша декларуватиме очевидний факт, що при сортуванні порожнього списку виходить порожній список. Друга затверджує, що одноелементний список також вже є впорядкованим. В третьому правилі міститиметься суть методу сортування злиттям. Спочатку список розщеплюється на два підсписки за допомогою предиката splitting, потім кожний з них сортується рекурсивним викликом предиката fusion sort, і, нарешті, використовуючи предикат fusion, з’єднати отримані впорядковані підсписки в один список, який і буде результатом впорядковування елементів початкового списку.

Фактично цей алгоритм при прямому проході дробить список на одноелементні підсписки, після чого на зворотному ходу рекурсії зливає їх двоелементні списки. На наступному етапі зливаються двоелементні списки і т. д. На останньому кроці два підсписки зливаються в підсумковий, відсортований список.

Далі створений предикат, який перевірятиме, чи є список впорядкованим. Це зовсім не складно. Для того, щоб список був впорядкованим, він повинен

бути або порожнім, або одноелементним, або будь-які два його сусідні елементи повинні бути розташовано в правильному порядку. Запишемо ці міркування:

```
domains
list=integer*

predicates
поділити(list,list,list)
переміст_сорт(list,list)
сортувати(list)
перемістити(list,list,list)

clauses
поділити([],[],[]).      /* порожній список  можна розділити тільки на порожні
                            підсписки */
поділити([H],[H],[]). /* одноелементний список розбити на одноелементний
                            список та порожній список */
поділити([H1,H2|T],[H1|T1],[H2|T2]):-
 поділити(T,T1,T2).
                          /* елемент H1 відправляємо в перший підсписок,
                            елемент H2 - в другий підсписок, хвіст T розбиваємо
                            на підсписки T1 и T2 */
перемістити([ ],[ ],[ ]).
переміст_сорт([],[]):-!./*  відсортований порожній список залишається
                              порожнім списком */

переміст_сорт([H],[H]):-!. /* одноелементний список впорядкований */
переміст_сорт(L,L_s):-
                    поділити(L,L1,L2),
                              /* розділяємо список L на два підсписка */
                    переміст_сорт(L1,L1_s),
                              /* L1_s - результат сортування L1 */
                    переміст_сорт(L2,L2_s),
                              /* L2_s - результат сортування L2 */
                    перемістити(L1_s,L2_s,L_s).
                              /* з'єднуємо L1_s и L2_s в список L_s */
сортувати([ ]).             /* порожній список відсортовано */
сортувати([_]).             /* одноелементний список впорядковано */
сортувати([X,Y|T]):-
                    X<=Y,
                    сортувати([Y|T]).
                              /* список впорядкований, якщо перші два елемента
                              розташовані в правильному порядку та список,
                              створений другим елементом та хвостом
                              вхідного, впорядковано */
goal
поділити([5,1,7,2,9,4,8],L1_s,L2_s).
```

Відповідь Пролог:

```
L1_s=[5,7,9,8], L2_s=[1,2,4]
1 Solution
```

Приклад. Обчислення суми елементів списку.

Алгоритм обчислення суми елементів списку аналогічний підрахунку кількості елементів списку. Різниця між ними полягає в реалізації кроку рекурсії: при підрахунку кількості елементів неважливо, чому рівний перший елемент списку, слід просто додавати одиницю до вже підрахованої кількості елементів хвоста; при обчисленні суми – слід враховувати значення голови списку.

Сума елементів порожнього списку рівна нулю. Для обчислення суми елементів не порожнього списку потрібно до суми елементів хвоста додати перший елемент списку:

```
sum([ ],0).        % сума елементів порожнього списку рівна нулю
sum([H|T],S):-
sum(T,ST)      % ST - сума елементів хвоста */
S=ST+H.        % S - сума елементів початкового списку */
```

Інший варіант обчислення суми елементів списку – збереження проміжного значення обчисленої суми (накопичувач). На початку сума дорівнює нулю – накопичувач порожній. Переходячи від обчислення суми непорожнього списку до обчислення суми елементів його хвоста, слід додавати перший елемент до вже обчисленої суми. Коли елементи списку будуть вичерпані (список спустіє), накопичена сума передається як результат в третій аргумент:

```
Sum_list([ ],S,S).
/* список став порожнім - значить сума
елементів списку в накопичувачі */

sum_list([H|T],N,S):-
N=H+N
/* NT - результат додавання до суми, що знаходиться в
накопичувачі, першого елемента списку */
sum_list(T,N T,S).
/* виклик предикат від хвоста T і N T */
```

Якщо потрібно викликати предикат не від трьох аргументів, а від двох, то можна додати допоміжний предикат:

```
sum2(L,S):–
sum_list(L,0,S).
```

Останній варіант, на відміну від першого, реалізує хвостову рекурсію.

Приклад. Середнє арифметичне елементів списку.

Простіше скористатися предикатами, які дозволяють обчислити кількість і суму елементів списку. Для знаходження середнього арифметичного елементів списку достатньо буде суму елементів списку розділити на їх кількість:

```
avg(L,A):–
summa(L,S) % помістити в змінну S суму елементів списку
length(L,K)   % змінна K рівна кількості елементів списку
A=S/K.        % середнє арифметичне – відношення суми до кількості
```

Проблема виникає при спробі знайти середнє арифметичне елементів порожнього списку. Якщо спробувати викликати ціль *avg([ ],A)*, то результатом виконання буде повідомлення про помилку *"Division zero"* («Ділення на нуль»). Це відбудеться, тому що предикат *length([],K)* конкретизує змінну K нулем, а при спробі досягнення третьої підцілі *A=S/K* і відбудеться вищезазначена помилка. Можна вважати це нормальною реакцією предиката. Раз в порожньому списку немає елементів, значить, немає і їх середнього арифметичного. А можна змінити цей предикат так, щоб він працював і з порожнім списком.

Щоб усунути проблему з порожнім списком, слід додати в процедуру, у вигляді факту, інформацію про те, що середнє арифметичне елементів порожнього списку дорівнює нулю. Повне рішення виглядатиме наступним чином:

```
avg([ ],0):–!.

avg(L,A):-
summa(L,S),
length(L,K),
A=S/K.
```

Описуючи цей предикат в розділі опису предикатів, слід звернути увагу на те, що другий аргумент вже буде не цілого типу.

Приклад. Предикат, що знаходить мінімальний елемент списку.

Рішення буде рекурсивним, але оскільки для порожнього списку поняття мінімального елемента не має сенсу, базис рекурсії слід записати не для порожнього, а для одноелементного списку. В одноелементному списку, природно, мінімальним елементом буде той самий єдиний елемент списку.

Реалізація кроку рекурсії: знайти мінімум з першого елемента списку і мінімального елемента хвоста – це і буде мінімальний елемент всього списку:

```
min_list([X],X).
/* єдиний елемент одноелементного списку є
мінімальним елементом списку */
min_list([H|T],M):-
min_list(T,MT)
/* M T - мінімальний елемент хвоста */
min(H,MT,M).

/*M - мінімум з MT і першого елемента початкового списку */
```

Злегка модифікувавши предикат *min_list* (підставивши в правило замість предиката *min* предикат max і помінявши його назву), отримаємо предикат, що знаходить не мінімальний, а максимальний елемент списку.

Приклад. Написати програму, що створює список міст. Виконати програму з різними внутрішніми і зовнішніми цілями.

```
domains
town_list=town *
town=symbol

predicates
towns(town_list)
run()

clauses
towns(["Київ","Дніпропетровськ","Львів","Харків","Полтава"]).
run:-towns([A,B,C,D,E]),write(A,",", B,",", C,",", D,",", E,".").
run.

goal
run.
/*Результат:
Київ, Дніпропетровськ, Львів, Харків, Полтава yes*/
```

Або цей же приклад:

```
domains
town_list=town*
town=symbol

predicates
towns(town_list)

goal
towns([A,B,C,D,E]),write (A,",", B,",", C,",", D,",", E,". "),nl.

clauses
towns(["Київ", "Дніпропетровськ", "Львів", "Харків", "Полтава"]).
```

```
/*Результат:
Київ,Дніпропетровськ,Львів,Харків,Полтава.
A=Київ, B=Дніпропетровськ, C=Львів, D=Харків, E=Полтава
1 Solution*/
```

Запитання. Завдання

1. Визначення списку у Visual Prolog?
2. Чи можуть бути елементи списку різних типів?
3. Як оголошується список?
4. Складова структура одноелементного списку?
5. Спосіб обробки списку?
6. Алгоритм обробки списку?
7. Як визначається довжина списку?
8. Що є головою списку?
9. Дії, які можна виконувати зі списками?
10. Методи сортування елементів списку.

Тест для самоконтролю знань з розділу 7[F]

1. Список у Пролозі – це ...?
   а) предикат, аргументами якого є складені структури даних;
   б) об’єкт, який містить кінечне число інших об’єктів;
   в) рекурсивна структура даних.
2. Складовою списку є ...?
   а) елемент;
   б) аргумент;
   в) об’єкт.
3. Як оголошується список?
   а) доменом (list*) в розділі доменів, наприклад, list*=string
   б) доменом (list) в розділі доменів, наприклад, list=string
   в) символом (*) в розділі доменів після оголошення типу елементів списку, наприклад, list=string*
4. Прокоментуйте наступну декларацію у розділі доменів:

```
elementList=elements*
elements=i(integer); r(real); s(symbol)
```

   а)декларація елементів списку різних типів через домен користувача elements;
   б) декларація елементів списку, що належать до одного стандартного домену;
   в) декларація багаторівневої структури даних.

[F] Усі запитання мають один правильний варіант відповіді.

5. Список складається з …?
   а) голови і хвоста, які можуть бути як елементом списку так і самим списком;
   б) голови, яка є елементом або списком, та хвоста, який є завжди списком, включаючим всі наступні елементи;
   в) голови, яка є першим елементом, і хвоста, який є завжди списком, що включає всі наступні елементи.
6. Який список не можна поділити на голову і хвіст?
   а) елементи якого самі є списком;
   б) деякі елементи якого можуть представляти собою список;
   в) порожній список;
   г) всі відповіді вірні.
7. Вкажіть відповідь з синтаксично правильним записом списку:
   а) [a, b, c]
   б) [a| [b, c]]
   в) [a| [b| [c]]]
   г) всі відповіді вірні.
8. Які роздільники застосовуються для структурного відокремлення голови і хвоста списку?
   а) прямокутні дужки;
   б) коми;
   в) вертикальні риски;
   г) всі відповіді вірні.
9. У чому полягає робота зі списком?
   а) у рекурсивному відокремленні його голови до тих пір, поки список не стане порожнім;
   б) у перевірці, чи міститься елемент у списку, і чи включає список в себе інший список.
10. Як оголошуються елементи списку:
   а) елементи списку можуть бути оголошені різними стандартними доменами;
   б) елементи списку можуть бути оголошені з використанням функторів для різних доменів;
   в) всі відповіді вірні.

РОЗДІЛ 8. СИМВОЛЬНІ РЯДКИ

Рядок визначається як список кодів символів. Коди символів мають особливе значення в мовах програмування. Вони виступають як засіб зв'язку комп'ютера із зовнішнім світом. У Пролозі існує спеціальний синтаксис для запису символьних рядків, подібний до синтаксису атомів. Символьним рядком (або просто рядком) є будь-яка послідовність символів, які можуть бути надруковані (крім подвійних лапок), вкладена в подвійні лапки. Подвійні лапки в межах символьного рядка записуються двічі "".

Рядки розглядаються як певний тип об'єктів подібно атомам або списками, тому поняття рядка є чисто синтаксичним. Специфіка обробки рядків у Visual Prolog полягає у використанні деяких вбудованих предикатів, призначених для роботи з рядками. Ці предикати можна розділити на дві групи:

- базові предикати управління рядками;
- предикати для перетворення рядків в інші типи і навпаки.

8.1. Предикати управління рядками

Дані предикати являються основою обробки рядків і виконують наступні дії:

- перевірка/визначення довжини рядка;
- створення порожнього рядка визначеної довжини;
- поділ рядків на підрядки і лексеми;
- побудова рядків з визначених підрядків і лексем;
- перевірка, чи складається рядок з визначених підрядків чи лексем;
- повернення лексеми або списку лекскм з даного рядка;
- перевірка, чи є рядок терміном, визначеним у Visual Prolog;
- форматування змінної кількості аргументів у рядкову змінну.

У програмуванні *лексема* – це послідовність машинних символів вихідного коду програми, що мають певне сукупне значення. Для більшості мов програмування актуальні такі класи лексем:

- зарезервовані слова;
- ідентифікатори;

- числові константи (цілі та дійсні числа);
- літерні константи;
- рядкові константи;
- коди операторів;
- коментарі;
- дужки та інші елементи програми.

В рядках Visual Prolog зворотний слеш «\» являється управляючим символом, який дозволяє вставляти в рядки символи, відсутні на клавіатурі.

Предикат str_len

Вбудований предикат *str_len* призначений для визначення довжини рядка, тобто кількості символів, що входять в рядок. Він має два аргументи: перший – рядок, другий – кількість символів:

```
str_len(StringsArg,Langth)
```

Є три варіанти використовування даного предиката.

Перший, найприродніший варіант використовування цього предиката, коли перший аргумент зв'язаний, а другий вільний – *(i,o)*. У цьому випадку в другий аргумент буде поміщено кількість символів, яку містить перший аргумент.

Другий варіант – коли обидва аргументи зв'язані *(i,i)*. У цьому випадку предикат буде успішний, якщо довжина першого аргументу співпадатиме з другим аргументом, інакше – неуспішний.

Рідше використовується третій варіант, коли другий аргумент зв'язаний, а перший – вільний *(o,i)*. В цій ситуації перший аргумент буде зазначений рядком, що складається з пропусків, причому кількість пропусків буде рівна другому аргументу.

Наприклад, для цільового твердження:

```
goal
str_len("абвг",K).
```

результатом є *K=4*.

Для цілі:

```
goal
```

```
str_len("абвг",4).
```

результатом є підтвердження факту *Yes*

А для цілі:

```
goal
str_len(STR,15),writef("|%|\n",STR).
```

результат

```
|   |
STR=
1 Solution
```

Предикат concat

Призначений для з'єднання двох рядків – конкатенації. Предикат детермінований. У нього три аргументи, кожний рядкового типу, принаймні, два з трьох аргументів повинні бути зв'язаними:

```
Concat(String1,String2,String3)
```

Можливі чотири шаблони або чотири варіанти використовування цього предиката: *(i,i,o)*, *(i,o,i), (o,i,i), (i,i,i)*.

Перший варіант, коли зв'язано перші два аргументи. В цьому випадку третій аргумент буде зазначений рядком, отриманим приписуванням другого аргументу до першого *(i,i,o)*.

Другий варіант *(i,o,i)*, коли зв'язаними виявилися перший і третій аргументи. В цій ситуації другий аргумент буде зазначений рядком, при приписуванні якого до першого аргументу вийде третій аргумент (якщо, звичайно, такий рядок існує). Якщо такого рядка не може бути, предикат буде неуспішний.

Третій варіант аналогічний другому, за винятком того, що зв'язаними аргументами виявляються другій і третій, а вільним – перший *(o,i,i)*. В цьому випадку, природно, перший аргумент буде зазначений рядком, при приписуванні до якого можна отримати третій аргумент, якщо це взагалі можливо. Інакше предикат терпить невдачу.

Четвертий варіант *(i,i,i)* виникає, коли всі три аргументи визначено *(i,i,i)*. Предикат буде успішний, якщо при з'єднанні першого аргументу з другим вийде третій аргумент, інакше – неуспішний.

Наприклад, для цільового твердження:

```
goal
concat("ааа","ббб",X).  /*(i,i,o)*/
```

пролог виведе результат *X=аааббб*

Для цілі:

```
goal
concat("ааа",X,"ааа----ббб")./*(i,o,i)*/
```

результат *X=----ббб*

У випадку наступного цільового твердження:

```
goal
concat(X,"ббб","ааа----ббб"). /*(o,i,i)*/
```

результатом є *X=ааа----*

У випадку:

```
goal
concat("ааа","ббб","аааббб").       /*(i,i,i)*/
```

результатом є підтвердження *Yes*.

І у останньому випадку:

```
goal
concat("ааа","ббб","ааа----ббб").   /*(i,i,i)*/
```

відповідь *No*.

Предикат frontchar

Використовується для розділення початкового рядка на перший символ і «хвіст», що складається з символів, які залишилися після видалення першого символу. Це нагадує представлення списку у вигляді голови і хвоста. Причому перший і третій аргументи даного предиката належать рядковому домену, а другий – символьному:

```
frontchar(String1, Char, String2)
```

У цього предиката п’ять варіантів використовування:

(i,o,o), *(o,i,i)*, *(i,i,o)*, *(i,o,i)*, *(i,i,i)*

Перший варіант *(i,o,o)*, коли перший аргумент зв’язаний, а другий і третій – вільні. В цьому випадку другий аргумент буде зазначений першим символом рядка, а в третій аргумент будуть записані всі символи початкового рядка, починаючи з другого.

Другий *(o,i,i)*, також часто використовуваний варіант цього предиката, коли навпаки, перший аргумент вільний, а другий і третій – є зв'язаний. В цьому випадку в перший аргумент потрапить рядок, виконаний приписуванням рядка, що знаходиться в третьому аргументі, до символу, який зазначений другим аргументом. Це аналог конкатенації, але з'єднуються не два рядки, а символ і рядок.

Третій варіант *(i,i,o)* виходить, коли перший і другий аргументи визначені, а третій відсутній. В цій ситуації в третій аргумент будуть переписані всі символи першого аргументу, починаючи з другого, у випадку, якщо перший символ першого аргументу співпадає з другим аргументом. Інакше предикат терпить невдачу.

Четвертий варіант *(i,o,i)* схожий на третій, але вільний другий аргумент, а зв'язаним є перший і третій. В цьому випадку другий аргумент визначається першим символом першого аргументу, якщо символи першого аргументу, що залишилися, утворюють в точності третій аргумент. Інакше предикат буде неуспішний.

П'ятий варіант використовування цього предиката виникає, коли всі три його аргументи визначені *(i,i,i)*. В цьому випадку він буде істинний, якщо рядок, що зберігається в першому аргументі, співпадає з рядком, отриманим приписуванням до символу з другого аргументу рядка з третього аргументу. Інакше предикат буде помилковий.

Приклади застосування предиката *frontchar*.

Для цілі:

```
goal
frontchar("VPROLOG",Символ,Решта).
```

рішення Пролог *Символ =V, Решта =PROLOG*.

Для цілі:

```
goal
frontchar("VPROLOG",'V', Решта).
```

рішення *Решта =PROLOG*.

Для цілі:

```
goal
frontchar("VPROLOG",'P',"ROLOG").
```

рішення *No*.

Для цілі:

```
goal
frontchar(Сум_рядок,'V',"PROLOG").
```

рішення *Сум_рядок =VPROLOG*.

Предикат frontstr

Предикат *frontstr* узагальнює предикат *frontchar* в тому значенні, що він теж дозволяє відокремити від даного рядка деяку кількість символів, але не обов'язково один.

```
frontstr (NumberOfChars, SrcString, StartStr, EndString)
```

Предикат має чотири параметри *(i i,o,o)*.

У першому параметрі указується кількість символів (*unsigned*), які копіюються з другого параметра в третій, залишком другого параметра є четвертий аргумент. Цей предикат може використовуватися тільки єдиним описаним вище способом, а саме, перші два параметри у нього вхідні, а третій і четвертий – вихідні.

Наприклад, на цільове твердження:

```
goal
frontstr(9,"Програмне забезпечення АС",STR1,STR2).
```

Пролог виведе рішення:

```
STR1=Програмне,STR2=забезпечення АС
1 Solution
```

А у випадку:

```
goal
frontstr(0,"Програмне забезпечення АС",STR1,STR2).
```

Пролог знайде:

```
STR1= ,  STR2=Програмне забезпечення АС.
```

Якщо кількість символів, вказаних в першому параметрі предиката *frontstr*, перевищує довжину рядка з другого параметра, предикат терпить невдачу.

Предикат fronttoken

fronttoken призначений для поділу рядка на лексему та залишок – дробить початковий рядок, вказаний як перший параметр предиката, на дві частини. Формат:

fronttoken(String,Token,Rest)

В другий аргумент предиката потрапляє перший атом, що входить в рядок, розміщений в першому аргументі, в третій – залишок вхідного рядка, отриманий після видалення з нього атома (ідентифікатор, або беззнакове число, або символ).

У цього предиката існують п’ять варіантів використовування:

(i,o,o), *(o,i,i), (i,i,o), (i,o,i), (i,i,i)*

Перший варіант *(i,o,o)*, коли перший аргумент зв’язаний, а другий і третій – вільні. В цьому випадку другий аргумент буде зазначений першим атомом рядка, що знаходиться в першому аргументі, а в третій аргумент буде записаний залишок початкового рядка.

Другий варіант *(o,i,i)* використовування цього предиката – коли, навпаки, перший аргумент вільний, а другий і третій – є зв’язаний. У такому разі виконується конкатенація рядків. В перший аргумент потрапить рядок, виконаний приписуванням рядка, що знаходиться в третьому аргументі, до рядка, який визначає другий аргумент.

Третій варіант *(i,i,o)* виходить, коли перший і другий аргументи визначені, а третій відсутній. Тепер в третій аргумент будуть переписані всі символи першого аргументу, що залишаються після видалення першого атома, у випадку, якщо перший атом першого аргументу співпадає з другим аргументом. Інакше предикат терпить невдачу.

Четвертий варіант *(i,o,i)* подібний третьому. В цьому випадку другий аргумент визначається першим атомом першого аргументу, якщо символи першого аргументу, що залишилися, співпадають з рядком, що знаходиться в третьому аргументі. Інакше предикат буде невдалий.

П’ятий варіант *(i,i,i)* виникає, коли всі три його аргументи визначені. В цьому випадку він буде істинним, якщо рядок, що зберігається в першому аргументі, співпадає з рядком, отриманим приписуванням до атома з другого аргументу рядка з третього аргументу. Інакше предикат буде помилковим.

Приклади застосування *fronttoken*.

Застосування *fronttoken* з параметрами *(i,o,o)*:

```
goal
fronttoken("Програмне забезпечення АС",Зн,Решта). /*(i,o,o)*/
```

результат *Зн=Програмне, Решта = забезпечення АС*.

Застосування *fronttoken* з параметрами *(i,i,o)*:

```
goal
fronttoken("копіювання+зчитування інформації",Зн,Решта),
fronttoken(Решта,Зн1,_).
```

результат виконання:

```
Зн=копіювання, Решта=+зчитування інформації, Зн1=+
1 Solution
```

Для цілі

```
goal
fronttoken("500Мбайт корисної інформації",Зн, Решта). /*(i,o,o)*/
```

результататом є

```
Зн=500, Решта =500Мбайт корисної інформації
1 Solution
```

Для цілі

```
goal
fronttoken("1.5Мбайт корисної інформації ",Зн, Решта). /*(i,o,o)*/
```

результат *Зн=1.5, Решта =5 Мбайт корисної інформації.*

Для цілі

```
goal
fronttoken("-1.5 мільйона років",TOK,REST). /*(i,o,o)*/
```

результат *Зн= -, Решта =1.5 мільйона років*.

Для цілі

```
goal
fronttoken(".55мільйона. років ",Зн, Решта).
```

результат *Зн=., Решта =5 мільйона. Років*.

Предикат isname

Перевіряє (*Yes* або *No*), чи являється аргумент допустимим іменем відповідно синтаксису Visual Prolog, тобто предикат істинний, якщо його рядковий аргумент є ідентифікатором (всі попередні чи наступні пропуски ігноруються), інакше помилковий. Ім'я розпочинається з символу алфавіту чи символу підкреслення, за яким слідують будь-які символи, цифри або символи підкреслення.

Формат запису предиката:

```
isname(String)
```

з одним вхідним параметром *(i)*.

Приклади застосування *isname*. Для наступної цілі:

```
goal
isname("ПРОЛОГ").
```

відповідь *Yes*.

У випадку такого цільового твердження:

```
goal
isname("1 ПРОЛОГ").
```

відповідь *No*.

Предикат frontstr

Призначений даний предикат для поділу рядка на дві частини. Формат запису наступний:

```
frontstr(NumberOfChars,String1,StartStrn,EndStr)
```

де:

String1 – рядок;

NumberOfChars – число перших символів з String1, які містить StartStrn;

EndStr – залишок символів з String1.

frontstr має параметри *(i,i,o,o)*, причому перші два з них завжди повинні бути зв'язаними, останні два – вільні.

Як приклад можна привести виконання Прологом наступної цілі, з першим параметром, що відрізняється.

Для цілі

```
goal
frontstr(0,"логічна мова програмування",STR1,STR2).
```

результатом є *STR1=, STR2= логічна мова програмування*.

Або для

```
goal
frontstr(12," логічна мова програмування ",STR1,STR2).
```

результат виконання: *STR1= логічна мова*, *STR2= програмування*.

Якщо задати ціль:

```
goal
frontstr(26," логічна мова програмування",STR1,STR2).
```

отримаємо *No Solution*, так як у фразі всього 26 символів, – значить, немає чого ділити.

Предикат format

Предикат format діє аналогічно предикату *writef*, але виводить результат у вигляді рядкової змінної. Format відповідає синтаксису:

format (STRING OutputString, STRING FormatString, Arg1, Arg2, ..., ArgN)

Потік параметрів предикату є *(o, i, i, i, i ...)*.

Arg1, Arg2, ..., ArgN мають бути константами або змінними, які належать до стандартних доменів. Рядок формату містить звичайні параметри, які друкуються без модифікації, і містяться у таблиці специфікації:

- параметр (необов'язковий) дефіса '-' вказує, що поле має бути таким, що вирівнюється по лівому краю. Інакше – по правому краю;
- номер десяткового дробу *m*, що конкретизує розмір(необов'язковий) мінімального поля;
- параметр *.p* – конкретизує або точність floating-point зображення, або максимальне число параметрів, які надруковані від рядка(необов'язковий);
- параметр *f* – конкретизує інший формат для цього об'єкту(необов'язковий).

Наприклад, в *f* полі, можна задати параметр, який говорить, що число відображається у фіксовано-десятковому зображенні (див. довідку Visual Prolog). Наприклад, для цілі:

```
goal
format(X,"this % the %'st % test",is,1,"small").
format(X,"Це % є %1'й % тест",,"нескладний").
```

результат виконання Пролог

```
X=це є 1'й нескладний тест
1 Solution.
```

Предикат subchar

Це предикат, який повертає символ даної позиції у рядку (перший символ у рядку має позицію 1). Предикат відповідає формату:

subchar (STRING String, UNSIGNED Position, CHAR RetChar)

з потоком параметрів *(i,i,o)*.

У випадку, коли другий аргумент (*Position*) вказує на неіснуючий символ, предикат *subchar* завершується невдало (*PROGRAM ERROR… String index error*), наприклад для даної цілі:

```
goal
subchar("ПРОЛОГ",10,Символ).
```

І повертає символ, коли ціль задана коректно:

```
goal
subchar("ПРОЛОГ",4,Символ).
```

Відповідь: *Символ=Л.*

Предикат substring

Предикат призначений для повернення частини рядка, містить чотири аргументи, виконується за форматом:

substring(STRING Source, UNSIGNED Pos, UNSIGNED Len, STRING Part)

з потоком параметрів *(i,i,i,o)* .

де: *Source* – рядок;

Pos – позиція символу;

Len – довжина рядка (копії);

Part – частина рядка *Source.*

Приклад виконання предиката *substring*:

```
goal
substring("Visual Prolog",8,3,Новий_рядок ).
```

Результатом наведеного цільового твердження є рішення, у якому Пролог виводить частину символів у змінній *Новий_рядок*:

```
Новий_рядок=Pro
1 Solution
```

Якщо параметри предиката вказані не коректно і не відповідають дійсності, то предикат завершується невдало. Але запит 0 байт в кінці рядка

```
substring("Visual Prolog",8,0,Новий_рядок).
```

помилкою не являється, змінна *Новий_рядок* зв'язується з порожнім рядком. Помилковим, наприклад, буде запис предиката у вигляді (або подібному):

```
goal
substring("Visual Prolog",14,1,Новий_рядок ).
```

Він викличе повідомлення про помилку.

Предикат searchchar

Цей предикат повертає позицію першої появи символу у рядку. Містить три аргумента з потоком параметрів *(i,i,o)* та відповідає формату:

```
searchchar(STRING String, CHAR SearchingChar, UNSIGNED Pos)
```

де: *String* – це вхідний рядок символів;

SearchingChar – символ рядка;

Pos – позиція вказаного символа.

Наприклад

```
goal
searchchar("ПРОЛОГ", 'О',Позиція).
```

має одне рішення

```
Позиція=3
1 Solution.
```

Недоліком є той факт, що у випадку, коли символи повторюються, пошук з поверненням для знаходження і відображення їх позиції не виконується. Предикат *searchchar* завжди буде виводити позицію вказаного символу, який зустрічається у рядку вперше. Для знаходження позиції наступного символу, що повторюється, треба скласти програму з використанням рекурсії:

```
predicates
nondeterm пошук_симв (string,char,integer)
nondeterm пошук_симв1(string,char,integer,integer)
nondeterm поз_симв(string,char,integer,integer,integer)

clauses
пошук_симв(Str,Ch,Pos):-
пошук_симв1(Str,Ch Pos,0).
```

```
пошук_симв1(Str,Ch,Pos,Old):-
searchchar(Str,Ch,Pos1),
поз_симв(Str,Ch,Pos,Pos1,Old).

поз_симв(_,_,Pos,Pos1,Old):-
Pos=Pos1+Old.
поз_симв(Str,Ch,Pos,Pos1,Old):-
frontstr(Pos1,Str,_,Rest),
Old1=Old+Pos1,
пошук_симв1(Rest,Ch,Pos,Old1).

goal
пошук_симв("ПРОЛОГ", 'О', P),write(P,'\n'),fail.
```

Тепер Пролог знайде всі можливі рішення:

```
3
5
No Solution
```

Предикат searchstring

Предикат searchstring застосовується Пролог у випадку, коли треба повернути позицію першої появи рядка символів у новому рядку. Його формат

searchstring (STRING SourceStr, STRING SearchStr, UNSIGNED Position)

Де два перших аргументи – це рядок (вхідний і новий) є вхідними параметрами, третій аргумент вказує позицію нового рядка і є вихідним параметром. Наприклад

```
goal
searchstring ("ПРОЛОГ", "ЛОГ", P).
```

Новий рядок розпочинається з 4 символу

```
P=4
1 Solution
```

Предикат буде неуспішний, коли фраза не знайдена.

Якщо вказана фраза повторюється декілька разів, треба написати програму, схожу для знаходження символів, що повторюються.

```
predicates
nondeterm пошук_ряд(string,string,integer)
nondeterm пошук_ряд1(string,string,integer,integer)
nondeterm поз_(string,string,integer,integer,integer)

clauses
пошук_ряд(Str,St,Pos):-
пошук_ряд1(Str,St,Pos,0).

пошук_ряд1(Str,St,Pos,Old):-
```

```
searchstring(Str,St,Pos1),
поз_(Str,St,Pos,Pos1,Old).

поз_(_,_,Pos,Pos1,Old):-
Pos=Pos1+Old.

поз_(Str,St,Pos,Pos1,Old):-
frontstr(Pos1,Str,_,Rest),
Old1=Old+Pos1,
пошук_ряд1(Rest,St,Pos,Old1).

goal
пошук_ряд("ПРОЛОГ ПРОЛОГ","ЛОГ",P),write(P,'\n'),fail.
```

Результат пошуку фрази, яка повторюється, є:

```
4
11
No Solution
```

8.2. Перетворення рядків в інші типи

Visual Prolog містить стандартні предикати, за допомогою яких можна виконувати перетворення типів будь-яких термів і рядків.

8.2.1. Предикати перетворення даних

Застосування предикатів перетворення даних доцільно, коли тип об’єкта вбудованого предиката відрізняється від типу об’єкта предиката, визначеного користувачем. Перетворення необхідні, коли значення одного типу повинно бути присвоєно до змінної іншого типу.

Предикат upper_lower

Виконує перетворення між верхнім і нижнім регістром літер. Формат:

upper_lower(рядок1,рядок2)

Аргументами є, або два рядки або два символи з потоком параметрів:

(i, o) – перетворює в нижній регістр еквівалент у верхньому регістрі.

(o, i) – перетворює в верхній регістр еквівалент в нижньому регістрі.

(i, i) – варіант потоку параметрів, успішний якщо два рядки (або символи) будуть рівні, якщо вони обидва були або повністю у верхньому регістрі або в нижньому регістрі. Іншими словами, *upper_lower (i,i)* забезпечує порівняння рядків з урахуванням регістра.

Приклади роботи предиката *upper_lower*. Для цілі:

```
goal
upper_lower("ЖОВТІ ВОДИ",Нижній).
```

як результат відображення фрази у нижньому регістрі *Нижній =жовті води*.

Або перетворення у верхній регістр:

```
goal
upper_lower(Верхній,"стратегія").
```

відповідь – *Верхній=СТРАТЕГІЯ*.

Порівняння рядків з урахуванням регістра:

```
goal
upper_lower("істИна","ІстИНа").
```

відповідь – *Yes*.

Предикат char_int

Призначений для перетворення символу в ціле число і навпаки. Формат:

char_int(char,integer)

(i,o) – повертає персональне число в ASCII коді для символу.

(o,i) – повертає символ, що відповідає числу в ASCII коді.

(i,i) – виконується, якщо число пов’язане в ASCII-код для символа, інакше не працює.

Приклади застосування *char_int*:

```
goal
char_int('a',X).
```

відповідь – *X=97*.

```
goal
char_int(X,97).
```

відповідь – *X=a*.

```
goal
char_int('a',97).
```

відповідь – *Yes*

```
goal
char_int('a',197).
```

відповідь – *No*.

Предикат str_char

Виконує перетворення рядка в символ і навпаки. Формат:

str_char(string, char)

аргументами якого є рядок і символ з наступними параметрами:

(o, i) – перетворює символ у рядок.

(i, o) – перетворює рядок у символ.

(i, i) повертає Істину, якщо обидва параметри пов'язані з уявленням одного і того ж характеру.

Наприклад, для цілі:

```
goal
str_char("A",Символ).
```

Пролог поверне рішення *Символ =A*.

Для цілі:

```
goal
str_char(Рядок,'A').
```

рішення – *Рядок=A*.

І нарешті третій варіант:

```
goal
str_char("A",'A').
```

Відповідь – *Yes*. А у випадку невідповідності аргументів, наприклад:

```
goal
str_char("ТОВАР",Символ).
```

рішень не знайдено. Або

```
goal
str_char("B",'A').
```

рішення – *No*.

Предикат str_int

Виконує перетворення рядка в ціле число і навпаки. Формат:

```
str_int(string,integer)
```

Параметри:

(i, o) – перетворює рядок у число.

(o, i) – перетворює число у рядок.

(i, i) – повертає Істину, якщо обидва параметри одного і того ж характеру.

Приклади використання предиката *str_int*:

```
goal
str_int("123",ЦілеЧисло).
```

відповідь *ЦілеЧисло=123*.

```
goal
str_int(Рядок,123).
```

відповідь *Рядок=123*.

```
goal
str_int("123",123).
```

відповідь *Yes*.

```
goal
str_int("-123",Число).
```

відповідь *Число=-123*.

```
goal
str_int("-1x23",Число).
```

відповідь *No Solution*.

Предикат str_real

Виконує перетворення рядка в дійсне число і навпаки. Формат:

```
str_real(string,real)
```

Предикат *str_real* виконується аналогічно *str_int*, наприклад:

```
goal
str_real("23456789079856",Real).
```

відповідь *Real=2.345678908E+13*.

```
goal
str_real(Str,-123234.453E-99).
```

відповідь *Str=-1.23234453E-94*.

```
goal
str_real("-1.23234453E-94",-123234.453E-99).
```

відповідь *Yes.*

8.2.2. Перетворення типів, що визначаються користувачем

Користувач може визначити перетворення, не передбачені предикатами. Наприклад, цілі числа в дійсні і навпаки:

```
predicates
conv_real_int(real,integer)

clauses
conv_real_int(R, N):-R =N.

goal
conv_real_int(2.5,N).
```

Результат перетворення:

```
N=3
1 Solution
```

Застосування розглянутих предикатів

Наступний приклад демонструє роботу предиката, який перетворюватиме рядок в список символів. Предикат матиме два аргументи. Першим аргументом буде даний рядок, другим – список, що складається з символів початкового рядка.

```
domains
список=char*

predicates
str_list(string,список)

clauses
str_list(" ",[ ]). /* порожньому рядку відповідає порожній список */

str_list(S,[H|T]):-
frontchar(S,H,S1),
/* H - перший символ рядка S, S1 - залишок рядка */
str_list(S1,T).

/* T - список, що складається  з символів, що входять
 в рядок S1*/

goal
str_list("цікава книга",X).
```

В результаті *frontchar* з фрази *"цікава книга"* поверне список символів:

```
X=['ц','і','к','а','в','а',' ','к','н','и','г','а']
1 Solution
```

Предикат, який перетворюватиме рядок в список атомів:

```
domains
список=string*

predicates
str_a_list(string,список)

clauses
str_a_list(" ",[ ]). % порожньому рядку відповідає порожній список

str_a_list(S,[H|T]):-
fronttoken(S,H,S1),
% H - перший атом рядка S, S1 - залишок рядка
str_a_list(S1,T).

% T – список  з атомів, що входять в рядок S1
```

```
goal
str_a_list("книга користується дуже великим попитом",X).
```

fronttoken перетворює фразу *"книга користується дуже великим попитом"* на список атомів:

```
X=["книга","користується","дуже","великим","попитом"]
1 Solution
```

Якщо ж в рядок можуть входити ще і розділові знаки, то кожний з них потрапить в підсумковий список окремим елементом. Для того, щоб отримати список, елементами якого є тільки слова, можна просто видалити з нього всі розділові знаки, що не складно.

Наступна програма демонструє предикат, який перетворює список символів в рядок. Предикат має два аргументи. Першим аргументом є список символів, другим – рядок, що складається з елементів списку.

Порожньому списку відповідає порожній рядок – базис рекурсії. Реалізація кроку: якщо початковий список не порожній, то потрібно перевести в рядок його хвіст, після чого, використовуючи стандартний предикат *frontchar*, приписувати до першого елемента списку рядок, отриманий з хвоста початкового списку.

```
domains
список=char*

predicates
        list_str(список,string)

clauses
list_str([ ]," "). /* порожньому рядку відповідає порожній список */

list_str([H|T],S):-
    list_str(T,S1),
   % S1 - рядок, освічений елементами списку T
frontchar(S,H,S1).
          /* S - рядок, отриманий  дописуванням рядка S1
          до першого елемента списку H */
goal
list_str(['я','с','к','р','а','в','е',' ','с','о','н','ц','е'],S).
```

frontchar перетворює список символів на рядок:

```
S=яскраве сонце
1 Solution
```

Використання предиката, який по рядку і символу підрахує кількість входжень цього символу в даний рядок. Предикат має три аргументи: перші два – вхідні (рядок і символ), третій – вихідний (кількість входжень другого аргументу в перший).

Рішення рекурсивне – ведеться підрахунок кількості входжень даного символу по рядку. Базис – якщо рядок порожній, то відповіддю буде нуль. Кроків рекурсії буде два залежно від того, чи буде першим символом рядка символ, входження якого треба знайти, чи ні. В першому випадку потрібно підрахувати, скільки разів шуканий символ зустрічається в залишку рядка, і збільшити отримане число на одиницю. В другому випадку (коли перший символ рядка відокремлений від даного символу, що рахується) збільшувати отримане число не потрібно. При розщеплюванні рядка на перший символ і хвіст потрібно скористатися предикатом *frontchar*.

```
predicates
   char_count(string,char,integer)

clauses
char_count(" ",_,0). /* будь-який символ не зустрічається
                        в порожньому рядку ні разу*/
char_count(S,C,N):-
frontchar(S,C,S1),!,
                     /* символ C виявився першим символом рядка S,
                     S1 – ті символи, що залишилися рядка S */
char_count(S1,C,N1),
                     /* N1 - кількість входжень символу C в рядок S1 */
N=N1+1.
                  /* N - кількість входжень символу C в рядок S виходить з
               кількості входжень символу C в рядок S1 додаванням одиниці */
char_count(S,C,N):-
frontchar(S,_,S1),
               /* першим символом рядка S  виявився символ, відмінний
               від початкового символу C, в S1 символи рядка S, що
               залишилися */
char_count(S1,C,N).
               /* в цьому випадку кількість  входжень символу C в рядок S
               співпадає з кількістю входжень символу C в рядок S1 */
goal
char_count("абабагаламага",'а',Кількість_а).
```

Результат пошуку Пролога:

```
Кількість_а=7
1 Solution
```

Наступна програма демонструє предикат, який по символу і рядку повертатиме першу позицію входження символу в рядок, якщо символ входить в рядок, і нуль, якщо не вихідний(позиція символа). У предиката три параметри. Перші два – вхідні (символ і рядок), третій – вихідний (перша позиція входження першого параметра в другий параметр або нуль).

Напочатку предикат *frontchar* розділить початковий рядок на перший символ і залишок рядка. Якщо першим символом рядка виявиться той самий символ, позицію якого треба знайти, тоді більше нічого робити не потрібно – Пролог поверне одиницю. В цьому випадку неважливо, які символи опинилися в хвості рядка, оскільки треба знайти перше входження даного символу в рядок.

Якщо першим символом початкового рядка є якийсь інший символ, відмінний від вказаного, то потрібно шукати позицію входження символу в залишок рядка. Якщо вказаний символ знайдеться в хвості, позиція входження символу в початковий рядок буде на одиницю більше, ніж позиція входження цього символу в залишок рядка. У всій решті ситуацій пошук вказаного символу не виконується – третій аргумент зазначається нулем.

```
predicates
str_pos(char,string,integer)

clauses
str_pos(C,S,1):-
frontchar(S,C,_),!.
% Вказаний символ C виявився першим символом даного рядка S

str_pos(C,S,N) :-
frontchar(S,_,S1),
/* S1 - складається зі всіх символів рядка S, окрім першого, який
відрізняється від вказаного символу C */
str_pos(C,S1,N1),
/* N1 - це позиція, в якій  символ C  зустрічається перший раз
в хвості S1 або ноль*/
N1<>0,!,
/* якщо позиція входження  символу C в рядок S1 не рівна
нулю, тобто якщо він зустрічається в рядку S1 */
N=N1+1.

/* то, збільшивши позицію його входження на одиницю,
отримаємо його позицію в початковому рядоку */
str_pos(_,_,0).  /* вказаний символ не входить в даний рядок */
```

```
goal
str_pos('р',"принтер",Позиція).
```

Рішенням Прологу перше входження символу в рядок є:

```
Позиція=2
1 Solution
```

Наступний приклад демонструє роботу предиката, який замінюватиме в рядку всі входження одного символу на інший символ. У предиката чотири параметри. Перші три – вхідні (початковий рядок, символ входження якого потрібно замінювати), символ (яким потрібно замінювати перший символ), вихідним четвертим параметром повинен бути результат заміни в першому параметрі всіх входжень другого параметра на третій параметр.

Якщо ж рядок непорожній, то задача полягає у тому, що треба розділити його за допомогою предиката frontchar на перший символ і рядок, що складається з решти символів початкового рядка.

Розглядаються два варіанти: або перший символ початкового рядка співпадає з тим, який потрібно замінювати, або не співпадає.

У першому випадку слід замінити всі входження першого символу другим символом в хвості початкового рядка, після чого, знову-таки за допомогою предиката frontchar, під’єднати отриманий рядок до другого символу. Як результ в результуючому рядку всі входження першого символу будуть замінені другим символом.

У другому випадку, коли перший символ початкового рядка не рівний символу, на який виконується заміна, слід замінити в хвості даного рядка всі входження першого символу на другий, після чого приєднати отриманий рядок до першого символу початкового рядка.

```
predicates
   str_replace(string,char,char,string)

clauses
 str_replace("",_,_,""):-!.
                        /* з порожнього рядка можна отримати тільки
                порожній
                        рядок */
 str_replace(S,C,C1,SO):-
 frontchar(S,C,S1),!,
                        /*символ С, що замінюється,виявився першим
```

```
                    символом рядка S, S1 - залишок рядка S */
str_replace(S1,C,C1,S2),
                    /*S2 - результат заміни в рядку S1 всіх входжень
                    символу C на символ C1 */
frontchar(SO,C1,S2).
                    /*SC - результат склеювання символу C1 і рядки S2 */
str_replace(S,C,C1,SO):-
frontchar(S,C2,S1),
          /*розділяємо початковий рядок S на перший символ C2 і
        рядок S1, заповнену всіма символами рядка S окрім першого */
str_replace(S1,C,C1,S2),
          /*S2 - результат заміни в рядку S1 всіх входжень символу C
                на символ C1 */
frontchar(SO,C1,S2).
          /* SO - результат з'єднання символу C1 і рядки S2 */
goal
str_replace("----------",'-','*',Рядок).
```

Результат заміни: *Рядок*=**********.

Для видалення частини рядка слід розробити предикат, який матиме чотири параметри: перші три вхідні (початковий рядок; позиція, з якої починається видалення; кількість символів, що видаляються), останній вихідний параметр – результат видалення з рядка.

Для цього слід за допомогою стандартного предиката *frontstr* поділити початковий рядок на два підрядки.

У другу частину входять всі символи, починаючи з вказаної позиції видалення. Другий підрядок треба ще раз розділити на два підрядки: в перший підрядок помістити ті символи, які потрібно видалити (в цьому місці можна буде скористатися анонімною змінною), в другий підрядок потраплять символи залишку початкового рядка, які не треба видаляти. Далі залишається тільки з’єднати перший підрядок початкового рядка з останнім підрядком другого підрядка, як наведено у наступному тексті програми:

```
predicates
 str_delete(string,integer,integer,string)

clauses
 str_delete(S,I,C,SO):-
 I1=I-1,       /* I1 - кількість символів, які повинні залишитися на початку
                  рядка S */
 frontstr(I1,S,S1,S2),
              /* S1 - перші I1 символів рядка S,
              S2 - символи  рядка S, з I -го до останнього */
```

```
 frontstr(C,S2,_,S3),
                /* S3 - останні символи рядка S2 ( або, останні символи
                      рядка S */
 concat(S1,S3,SO).
                /* SO - рядок, отриманий  з'єднанням рядків S1 і S3 */
goal
 str_delete("0123456789",5,3,РезультатПісляВидалення).
```

Після видалення залишився рядок з символів:

```
РезультатПісляВидалення=0123789
1 Solution
```

Наступна програма копіює частину рядка. Основний предикат складається з чотирьох параметрів: перші три вхідні (перший – є початковий рядок, другий – позиція, починаючи з якої потрібно копіювати символи, третій – кількість копійованих символів), четвертим вихідним параметром є результат копіювання символів з рядка. У програмі використовується предикат frontstr.

Спочатку треба розділити рядок, щоб отримати хвіст початкового рядка з тієї позиції, з якої потрібно копіювати символи.

Якщо після цього узяти стільки перших символів нового рядка, скільки їх потрібно копіювати, отримаємо в точності той підрядок початкового рядка, який вимагається отримати.

```
predicates
str_copy(string,integer,integer,string)

clauses
 str_copy(S,I,C,SO):-
 I1=I-1,   /* I1 - ця кількість символів,  розташованих на початку рядка S, які
              не потрібно копіювати */
 frontstr(I1,S,_,S1),
         % S1 - рядок, полягаючий зі всіх символів рядка S з I-го і до
останнього
 frontstr(C,S1,SO,_).
         /* SO - перші З символів рядки S1 */
goal
 str_copy("Копіювати рядок з позиції курсора",17,9,РезультатКопіювання).
```

Результат виконання *str_copy*:

```
РезультатКопіювання=з позиції
1 Solution
```

Наступний предикат реалізує розміщення одиного рядка всередині іншого. Предикат налічує чотири параметри: перші три вхідні (перший – це рядок, що вставляється; другий – рядок, в який потрібно вставити перший аргумент;

третій – позиція, починаючи з якої потрібно вставити перший параметр в другий), четвертим вихідним параметром буде результат вставки рядка.

Для реалізації цього предиката розділимо, використовуючи предикат frontstr, початковий рядок на два підрядки. В другу слід помістити всі символи, починаючи з позиції, в яку повинен бути вставлений другий рядок, в першу – початок початкового рядка, що залишився. Після цього, використовуючи конкатенацію, приписати до отриманого рядка той рядок, який потрібно було вставити. Для отримання остаточного результату залишається тільки дописати другий підрядок початкового рядка.

```
predicates
 str_insert(string,string,integer,string)

clauses
 str_insert(S,S1,I,SO):-
 I1=I-1,   /* I1 - ця кількість символів, розташованих на початку рядка S,
          після яких потрібно вставити нові символи */
 frontstr(I1,S1,S1_1,S1_2),
          /* S1 1 - перші I1 символів рядка S1, S1_ 2 -  залишок рядка S1, з I -го
          і до останнього */
 concat(S1_1,S,S2),
          /* S2 - рядок, отриманий об'єднанням рядків S1_ 1 і S */
 concat(S2,S1_2,SO).
          /* SO - рядок, отриманий злиттям рядків S2 і S1_ 2 */

goal
 str_insert(" з позиції","Вставляти рядок",10,Результат).
```

Як результат роботи програми отримаємо фразу:

```
Результат=Вставляти з позиції рядок

1 Solution
```

Наступний предикат дозволить пораховувати кількість цифр, якщо такі є в рядку, вхідним аргументом якого є рядок, вихідним – кількість цифр в рядку.

Основний предикат використовуватиме рекурсію по довжині рядка. Випадок, коли у рядку цифри відсутні є базисом рекурсії.

Для того, щоб реалізувати цей предикат, нам доведеться розробити допоміжний предикат, який буде означивать другий аргумент одиницею (якщо його перший аргумент є цифрою) і нулем (інакше).

Непорожній рядок за допомогою предиката frontchar слід розділити на перший символ і хвіст, підрахувати кількість цифр в хвості. Допоміжний предикат буде позначати 1 аргумент, якщо він є цифрою і 0, якщо це символ.

Після чого до отриманого числа слід додати одиницю, якщо перший символ – цифра, інакше – нуль.

```
predicates
 count_digit(string,integer)
 dig(integer,integer)

clauses
 dig(C,1):-'0'<=C,C<='9',!. % C - цифра
 dig(_,0).

 count_digit("",0):-!. % В порожньому рядку цифр немає

 count_digit(S,N):-
   frontchar(S,C,S2),
         % C - перший символ рядка S
         % S2 - хвіст рядка S
   dig(C,M), %M рівний одиниці, якщо C - цифра, і нулю, якщо інакше
   count_digit(S2,N2),
         % N2 - кількість цифр в рядку S2
   N=N2+M. /* Кількість цифр у всьому рядку більше на одиницю, чим
                кількість цифр у хвості, якщо перший символ рядка -
                цифра, і,інакше- рівно кількості цифр у хвості*/

goal
 count_digit("33 попуги та 1 крильце попуги",КількістьЦифр).
```

Результат пошуку та обчислення:

```
КількістьЦифр=3
1 Solution
```

Запитання. Завдання

1. Пояснити поняття «рядок» у Visual Prolog?
2. У чому полягає специфіка обробки рядків у Visual Prolog?
3. Дії, які виконують предикати обробки рядків?
4. Способи застосування предиката, призначеного для визначення довжини рядка?
5. Який вид детермінізму у предиката, призначеного для конкатенації рядків?

6. Який предикат представляє список у вигляді голови і хвоста, відокремлюючи перший (один) символ?
7. Який предикат призначений для поділу рядка на лексему та залишок?
8. У якому випадку використовується предикат frontstr?
9. Яка різниця між предикатом writef і format?
10. У якому випадку доцільно застосування предикатів перетворення типів даних будь-яких термів і рядків?

Тест для самоконтролю знань з розділу 8[F]

1. Які предикати використовує Пролог для обробки рядків?
   а) базові предикати;
   б) предикати перетворення рядкових типів;
   в) всі відповіді вірні.
2. У предикатів, призначених для розбиття рядка на компоненти чи побудови рядка із заданих компонентів, компонентами можуть бути:
   а) символи;
   б) лексеми;
   в).рядки;
   г) всі відповіді вірні.
3. Предикат str_len призначений для …
   а) перевірки довжини рядка;
   б) отримання довжини рядка;
   в) створення пустого рядка заданої довжини;
   г) всі відповіді вірні.
4. У якому випадку предикати frontstr, isname і format завершуються вдало?
   а) якщо варіанти потоку параметрів мають тільки вхідні параметри;
   б) якщо рядок, що розглядається, має задану довжину;
   в).якщо рядок, що розглядається, складається із заданих компонентів;
5. До предикатів перетворення типів належать …?
   а) frontchar, fronttoken, concat;
   б) frontstr, concat;
   в) char_int, str_int, upper_lower;
   г) всі відповіді вірні.
6. При якому шаблоні потоку параметрів предикати перетворення типів завершуються вдало?
   а) (i, o);
   б) (o, i) ;
   в). (i, i);
   г) (o, o).

[F] Усі запитання мають один правильний варіант відповіді.

7. Предикат, який виконує перевірку того, що рядок це є допустиме ім’я мови Visual Prolog?
   а) isname;
   б) frontchar;
   в) fronttoken;
   г) concat.
8. Що означає термін «лексема» у мові програмуванняVisual Prolog?
   а) це список аргументів типу string, що мають певне сукупне значення;
   б)це список аргументів типу symbol, що мають певне сукупне значення;
   в) це послідовність машинних символів вихідного коду програми, що мають певне сукупне значення;
   г) це коди операторів і коментарі.
9. У чому полягають особливості введення символу «\» у мові програмуванняVisual Prolog?
   а) це управляючий символ, який використовується для відображення шляху до об’єкту, наприклад D:\user\name1.pro
   б) це управляючий символ, який дозволяє вставляти в рядки символи, відсутні на клавіатурі;
   в) символ застосовується для відображення однорядкових коментарів.
10. Рядком може бути …
   а) будь-яка послідовність надрукованих символів;
   б) будь-яка послідовність надрукованих символів, крім круглих дужок, вкладена у подвійні лапки;
   в) будь-яка послідовність надрукованих символів, крім подвійних лапок, вкладена у подвійні лапки.

РОЗДІЛ 9. КЛАСИ І ОБ’ЄКТИ

Visual Prolog є повністю об’єктно-орієнтованою мовою.

Програма мовою Visual Prolog може бути написана як об’єктно-орієнтована програма з використанням класичних властивостей об’єктно-орієнтованої парадигми. Об’єктно-орієнтована система повинна задовольняти наступним критеріям:

- інкапсуляція;
- класи;
- спадкування;
- індивідуальність.

Інкапсуляція – це процес відокремлення один від одного окремих елементів об’єкта, що визначають його будову та поведінку. Інкапсуляція служить для того, щоб ізолювати інтерфейс об’єкта, що відображає його зовнішню поведінку, від внутрішньої реалізації об’єкта.

Об’єктний підхід припускає, що власні ресурси, якими можуть маніпулювати тільки методи самого класу, приховані від зовнішнього середовища. Абстрагування і інкапсуляція є взаємодоповнюючими операціями: абстрагування фокусує увагу на зовнішніх особливостях об’єкта, а інкапсуляція (або, інакше, обмеження доступу) не дозволяє об’єктам-користувачам розрізняти внутрішній склад об’єкта. Тобто, інкапсульовані об’єкти допомагають створювати більш структуровані та нескладні для читання програми.

Спадкування – це механізм об’єктно-орієнтованого програмування, який дозволяє побудову нового класу на основі вже існуючого (батьківського), з можливістю додавання чи перевизначення даних і методів. Це дозволяє поводитися з об’єктами класу-спадкоємця таким же чином, як з об’єктами базового класу. Нащадки успадковують характеристики батьківських класів без зміни їх первісного опису і додають при необхідності власні структури даних та методи.

Основними елементами об’єктної моделі Visual Prolog є *об'єкти*, *об’єктні типи* і *класи*.

9.1. Об’єкт

Об’єкт визначається як предмет або явище, що мають чітко визначену поведінку – набір іменованих об’єктів предикатів і набір підтримуємих інтерфейсів.

Об’єкт характеризується станом, поведінкою і індивідуальністю. Структура і поведінка схожих об’єктів визначають загальний для них клас. Терміни «екземпляр класу» і «об’єкт» є еквівалентними.

Стан об’єкта характеризується переліком усіх можливих (статичних) властивостей даного об’єкту і поточними значеннями (динамічними) кожної з цих властивостей.

Поведінка характеризує вплив об’єкта на інші об’єкти і навпаки щодо зміни стану цих об’єктів і передачі повідомлень. Інакше кажучи, поведінка об’єкта повністю визначається його діями.

Об’єкти не можуть бути одинаковими. Вони індивідуальні і визначаються своїм станом (атрибутами). *Індивідуальність* – це властивості об’єкта, що відрізняє його від усіх інших об’єктів.

Ієрархія класів у програмуванні означає класифікацію об’єктних типів, розглядаючи об’єкти як реалізацію класів (клас схожий на заготовку, а об’єкт – те, що будується на основі цієї заготовки) та пов’язуючи різні класи відносинами, подібно таким як «успадковує», «розширює», «є його абстракцією », «визначення інтерфейсу».

Основними засобами при роботі з об’єктами, об’єктними типами і класами є:

- інтерфейси (*interfaces*);
- оголошення класів (*class declarations*);
- виконання (*implementations*).

Інтерфейси – є об’єктні типи, вони описують доступ до об’єктів ззовні об’єкта. Інтерфейсом називається іменований набір оголошень предикатів, оформлений у вигляді розділу *interface*.

Наприклад визначення інтерфейсу з ім’ям *особа*:

```
interface особа

predicates
отриматиІм’я: ( ) - > string Ім’я.
встановитиІм’я: (string Ім’я).

end interface особа
```

У наведеному прикладі об’єкти типу *особа* мають два предиката *отриматиІм’я* і *встановитиІм’я*, які оголошуються в розділі *interface*.

Інтерфейси описують доступ до об’єктів ззовні об’єкта і являються об’єктними типами. Тільки класи, які мають інтерфейс, можуть бути використані як генератори об’єктів:

```
interface обробкаФайлу

predicates
видалитиЗФайлуРядок:(string Файл, unsigned ЧислоСимоволів)->string.

end interface обробкаФайлу
        /*декларація класу не містить нічого,
        крім покажчика на інтерфейс,
        що використовується,
        але її не можна виключити з коду*/

class обробкаФайлу: обробкаФайлу
end class обробкаФайлу

implement обробкаФайлу
видалитиЗФайлуРядок (Файл,ЧислоСимоволів)=ФрагментТексту:-
        ПовнийТекстФайлу=читатиФайл(Файл),
…
end implement обробкаФайлу
```

Ім’я інтерфейсу може не збігатися з ім’ям класу і, відповідно, імплементації:

```
interface робота_з_файлом

predicates
видалитиЗФайлуРядок:(string Файл, unsigned ЧислоСимоволів) ->string.
end interface робота_з_файлом

class обробкаФайлу: робота_з_файлом
end class обробкаФайлу

implement обробкаФайлу
видалитиЗФайлуРядок (Файл,ЧислоСимоволів)=ФрагментТексту:-
ПовнийТекстФайлу=читатиФайл(Файл),
...
```

```
end implement обробкаФайлу
```

Інтерфейси тільки визначають типи об’єктів, а класи породжують об’єкти.

9.2. Класи

Концепція класу у Visual Prolog заснована на наступних семантичних сутностях: об’єкти, інтерфейси і класи.

Клас – іменований предок об’єктів. Він може створювати об’єкти, що відповідають визначеним інтерфейсам. Будь-який об’єкт створюється за допомогою класу.

Клас – це множина об’єктів, пов’язаних спільністю структури і поведінки. Будь-який об’єкт є екземпляром класу. Визначення класів і об’єктів – одне з найскладніших завдань об’єктно-орієнтованого проектування.

В Visual Prolog текст програми поділено на окремі частини, кожна частина визначається як клас (*class*). В об’єктно-орієнтованих мовах програмування, *клас* – це пакет коду та асоційовані з ним дані. Клас можна уявляти собі як синонім поняття модуль (кожен клас зазвичай поміщається в окремий файл).

Кожен клас повинен мати декларацію класу та імплементацію. Декларація класу містить набір об’явлених констант, доменів, предикатів, видимих ззовні.

Реалізація класу використовується для визначення предикатів і конструкторів, оголошених в оголошенні класу, а також визначення будь-якого предиката при підтримці вбудованих предикатів.

Клас має розділ *class* оголошення (*declaration*) і розділ *implement* виконання.

Формат запису:

```
class <class_name> [: <base_class_list>]

  {[protected] domains <domains_declarations>}
  {[static] [protected] predicates  <predicates_declarations>}
  {[static] [protected] facts [- <facts_section_name>] <facts_declarations>}


endclass [<class_name>]
```

Наприклад, клас, який породжує об’єкт *особа*, можна оголосити таким чином:

```
class особа: особа
constructors
```

```
new: (string Ім’я).
end class особа
```

Це оголошення класу з ім’ям *особа*, що створює об’єкт типу *особа*. Клас має конструктор (*contructor*) з ім’ям *new*, який при виклику і породжує об’єкт типу *особа*, передаючи при цьому новому об’єкту рядок з ім’ям *Name*.

Нижче наведено код, коли декларація предиката і його клауза пов’язані однозначно, оскільки однозначно загальним ім’ям пов’язані декларація класу і його імплементація (виконання):

```
class обробкаФайлу

predicates
виділитиЗФайлуРядок:(string Файл, unsigned ЧислоСимоволів) ->string.

end class обробкаФайлу

implement обробкаФайлу
видалитиЗФайлуРядок (Файл,ЧислоСимоволів)=ФрагментТексту:-
                        ПовнийТекстФайлу=читатиФайл(Файл),
                        ...
end implement обробкаФайлу
```

Декларація класу – це сутність, невіддільна від імплементації класу з тим же ім’ям. Тобто, не можна в будь-яку довільну декларацію класу помістити декларацію предиката, а в якусь імплементацію класу з іншим ім’ям помістити клаузу і спробувати їх пов’язати посиланнями, що використовують ці імена. Ім’я класу однозначно визначає його декларацію і його імплементацію. Саме в них можна помістити декларацію предиката і його клаузу.

Код програми на Visual Prolog поділяється ключовими словами на секції різного виду шляхом використання ключових слів, що вказують компілятору, який код генерувати. Наприклад, є ключові слова, що позначають різницю між деклараціями і визначеннями предикатів і доменів. Зазвичай, кожній секції передує ключове слово. Ключових слів, що позначають закінчення секції, немає. Наявність іншого ключового слова позначає закінчення попередньої секції і початок іншої.

Винятком з цього правила є ключові слова *implement* і *end implement*. Код, що міститься між цими ключовими словами, є кодом, який відноситься до

конкретного класу (клас можна уявляти як модуль або розділ якоїсь програми більш високого рівня).

Область видимості конкретного класу лежить всередині кордонів, де клас визначений в інтервалі між ключовими словами *implement* – *end implement*. Предикати, визначені тут, можуть викликати один одного без кваліфікаторов класу та подвійної двокрапки (*::*). Visual Prolog розглядає код, поміщений між цими ключовими словами, як код, що належить одному класу. За ключовим словом *implement* обов’язково має слідувати ім’я класу.

В процесі виконання програми, часто буває так, що програмі може знадобитися викликати предикат, який визначений в іншому класі (файлі). Аналогічно, дані (константи) або домени можуть бути затребувані в іншому файлі.

Visual Prolog дозволяє робити такі взаємні посилання на предикати або дані, використовуючи так звану концепцію області видимості (*scope access*). Наприклад, є предикат з ім’ям *pred1* визначений у класі *class1* (який поміщається в іншому файлі, відповідно до стратегії середовища IDE), треба викликати цей предикат в тілі клауза деякого іншого предиката *pred2*, що знаходиться в іншому файлі (наприклад, *class2*). Тоді ось як предикат *pred1* мав би бути викликаний в тілі клауза предиката *pred2*:

```
clauses
pred3: -
...
!.

pred2: -
class1 :: pred1, /* pred1 невідомий в цьому файлі.
Він визначений в іншому файлі,
тому потрібний кваліфікатор класу.*/

pred3,
...
```

У наведеному прикладі видно, що тіло правила предиката *pred2* викликає два предиката *pred1* і *pred3*. Оскільки *pred1* визначено в іншому файлі (де визначено клас *class1*), остільки слово *class1* з символом *::* (дві двокрапки) передує слову *pred1*. Це можна назвати як кваліфікатор класу.

Але предикат *pred3* визначений всередині того ж самого файлу, що і предикат *pred2*. Тому тут немає необхідності використовувати *class2 ::* перед викликом предиката *pred3*.

Технічно така поведінка пояснюється наступним: область видимості (scope visibility) предиката *pred3* знаходиться всередині тієї ж області, де знаходиться предикат *pred2*. Тому тут не потрібно показувати, що *pred3* і *pred2* знаходяться в одному класі. Компілятор автоматично побачить визначення предиката *pred3* всередині тієї ж області, в класі *class2*.

Ключове слово *open* використовується для розширення області видимості класу. Воно повинно бути розміщено на початку коду класу, відразу після ключового слова *implement* (з ім’ям класу і, можливо ім’ям інтерфейсу) та інформувати компілятор про те, що у цей клас повинні бути додані імена (предикатів/констант/доменів), які були визначені в інших файлах.

```
open class1
...
clauses
pred3: -
...
!.

pred2: -
pred1,  /*кваліфікатор "class1 " більше не потрібен,
оскільки область видимості розширена
використанням ключового слова 'open' */

pred3,
...
```

У класі повинен бути розділ виконання (*implement*). Імплементація класу містить:

- власне реалізацію предикатів (клаузи), оголошених у декларації;
- декларації констант;
- декларації та реалізації предикатів, що використовуються в кляузах;
- декларації фактів;
- факти і предикати, оголошені в імплементації класу, ззовні не видимі.

Формат:

```
implement <class_name> [: <base_class_list>]

        {domains <domains_declarations>}
        {[static] facts <facts_declarations>}
        {[static] predicates <predicates_declarations>}
        {clauses <predicates_bodies>}

endclass [<class_name>]
```

Наприклад виконання для *особа*:

```
implement особа

facts
імя:string.

clauses
new(Імя):-
імя:=Імя.

clauses
отриматиІмя()=імя.

clauses
встановитиІмя(Імя):-
                    імя:=Імя.
end implement особа
```

Як видно, в цьому розділі повинні бути присутніми визначення для кожного з видимих ззовні (*public*) предикатів і конструкторів, тобто для *new*, *отриматиІмя* і *встановитиІмя*.

У розділі *implement* можна, крім того, оголосити і визначити додаткові локальні елементи, доступ до яких можливий тільки з самого розділу *implement*. Так, клас містить оголошення факту-змінної *імя*, який буде використовуватися для зберігання імені персони.

Кожен з об’єктів при створенні отримує свою власну реалізацію факту-змінної. Вище наведені клаузи будуть посилатися тільки на конкретну реалізацію факту-змінної всередині об’єкта. Цей предикат є об’єктним предикатом (*object predicate*), а факт є об’єктним фактом (*object fact*).

9.3. Динамічні і статичні класи

Якщо клас має інтерфейс, то він є *динамічний*, інакше – *статичний*.

Динамічний клас – клас, здатний породжувати об'єкти. Динамічні класи можуть містити як статичні, так і динамічні предикати, а також статичні і динамічні факти. Кожен екземпляр динамічного класу містить свою копію динамічного предиката і свою копію динамічного розділу фактів. Створення об’єкта здійснюється викликом предиката-конструктора.

Статичний клас – клас, який не здатний породжувати об’єкти. Всі предикати і розділи фактів статичного класу існують в єдиному екземплярі у всьому проекті.

Наприклад, клас *staticClass* є статичним классом:

```
class staticClass
end class staticClass

implement staticClass
end implement staticClass
```

Але цей клас нічого не здатний робити, тому в нього слід додати декларації предикатів і клаузи:

```
class staticClass

predicates
visibleStaticPred:().
end class staticClass

implement staticClass

clauses
visibleStaticPred():-
…,
 invisibleStaticPred( ),
… .

class predicates
invisibleStaticPred:().

clauses
invisibleStaticPred():-
…
end implement staticClass
```

Звернення до предикатів цього класу записуються як:

```
...
staticClass :: staticVisiblePred (),
/* Істотно використання саме символу :: для звернення до статичної сутності.*/
…
```

Тут предикат *invisibleStaticPred* є предикатом, оголошеним всередині імплементації, і тільки всередині цієї імплементації може бути викликаний. Тобто предикати, оголошені в декларації класу, є свідомо завжди статичними предикатами, а розділ оголошень предикатів всередині імплементації статичного класу завжди повинен мати вид *class predicates*. Якщо спробувати створити екземпляр класу *staticClass* за допомогою виклику конструктора *new()*

```
...
staticClass::new(),
...
```

то це не пропустить компілятор. Таким чином, статичні класи – це просто модулі проекту. Проект, побудований лише на модулях (статичних класах) – не має ніякого відношення до об’єктно-орієнтованої концепції програмування.

Наприклад, клас *dynamicClass* є динамічним класом, що визначається конструкцією *dynamicClass: justInterface*.

```
class dynamicClass:justInterface
end class dynamicClass

implement dynamicClass
end implement dynamicClass

interface justInterface
end interface justInterface
```

Синтаксично все правильно, але практично марно. Тепер додамо корисності і винесемо на початок декларацію інтерфейсу:

```
interface justInterface

predicates
visibleDynamicPred:().

end interface justInterface

class dynamicClass:justInterface
predicates
visibleStaticPred:().
end class dynamicClass

implement dynamicClass

clauses
visibleStaticPred():-

invisibleStaticPred(),
```

```
...
clauses
visibleDynamicPred():-

nvisibleStaticPred(),
...
class predicates
invisibleStaticPred:().

clauses
nvisibleStaticPred():-

predicates
nvisibleDynamicPred:().

clauses
nvisibleDynamicPred():-

end implement dynamicClass
```

Маємо чотири предиката, які мають різні динамічні особливості. *visibleDynamicPred* – динамічний предикат, що належить интерфесу *justInterface*, видимий ззовні. *visibleStaticPred* – статичний предикат, видимий ззовні. *invisibleStaticPred* – статичний предикат, визначений в імплементації класу *dynamicClass*, невидимий ззовні. *invisibleDynamicPred* – динамічний предикат, визначений в імплементації класу *dynamicClass*, невидимий ззовні. До такого класу можна звернутися як до статичного:

```
...
dynamicClass::visibleStaticPred(),
...
```

але так не можна:

```
...
dynamicClass::visibleDynamicPred(),
...
```

Щоб викликати предикат *visibleDynamicPred*(), необхідно за допомогою конструктора створити екземпляр класу і звернутися до цього екземпляру:

```
...
екземплярDynamicClass=dynamicClass::new(),
екземплярDynamicClass:visibleDynamicPred(),
...
```

9.4. Факти, константи і домени

Факти декларуються і можуть бути використані в клаузах предикатів тільки в імплементації класу. Не можна оголосити факти в одному класі і намагатися використовувати звернення до них в іншому класі. У статичних класах факти повинні бути оголошені тільки як статичні.

```
implement example
...
class facts
myFact_F: (string, string).
...
end implement example
```

У імплементації динамічних класів факти можуть бути оголошені або як статичні, або як динамічні:

```
class example1:example1interface

predicates
записатиМісто:(string Місто).
записатиІмяАдреса:(string Імя, string Адреса).
отриматиМісто:()->string Місто.
отриматиІмяАдреса:(string Імя, string Адреса).

end class example1

implement example1
...
class facts
місто_V:string Місто.
...
facts
імяАдреса_F:(string Адреса, string Імя).

clauses
записатиМісто (Місто):-
місто_V:= Місто.
...
clauses
записатиІмяАдреса(Імя,Адреса):-
assert(імяАдреса_F:(Адреса,Імя)).

clauses
отриматиМісто (місто_V).
...
clauses
отриматиІмяАдреса(Імя,Адреса):-
імяАдреса_F(Адреса,Імя),
!,
...
```

```
end implement example1
```

У наведеному прикладі класу *example1* факт *місто_V* є статичним, а факти *імяАдреса_F(...)* є динамічними, що визначається різницею в імені розділу *class facts* і просто *facts*. Оскільки статична сутність будь-якого класу всього одна, а динамічна – по одній на кожен экземпляр, то в прикладі запит *отриматиМісто( ),* переданий будь-якому екземпляру, поверне один і той же результат, а запит *отриматиІмяАдреса(...)* – поверне для кожного екземпляра своє значення.

У прикладі використовувався *факт-змінна місто_V*, особливістю якого є те, що такий факт завжди один і він сам несе значення терма, оголошеного в декларації.

```
class facts
місто_V:string Місто.
```

Допускаються будь-які значення оголошених доменів користувача, наприклад:

```
domains
мійДомен=персона(string Імя,unsigned Вік).
class facts
персона_V: мійДомен.
```

Факт-змінна завжди існує і повинна мати початкове значення, яке можна привласнити при оголошенні:

```
domains
мійДомен=персона(string Імя,unsigned Вік).

class facts
персона_V:мійДомен:=персона("Людина похилого віку",80).
місто_V:string:="Відомості відсутні".
```

В динамічних класах початкове значення можна встановити у клаузі конструктора:

```
implement dynamicClass

domains
мійДомен=персона(string Імя,unsigned Вік).

class facts
персона_V:мійДомен.

clauses
new():-
персона_V:=персона("Літня людина",80),
```

```
...
end implement dynamicClass
```

У випадках, коли початкове значення факт-змінної неможливо встановити за сенсом реалізації, тоді воно може бути як «не встановлене»:

```
class facts
місто_V:string:=erroneous. % erroneous – ключове слово мови - помилка
```

За допомогою детермінованого предиката *isErroneous(...)* можна перевірити, чи являється значення факту не встановленим:

```
implement dynamicClass

domains
мійДомен=персона(string Імя,unsigned Вік).

class facts
місто_V:string:=erroneous.
персона_V:мійДомен.

clauses
new():-
персона_V:=персона("Літня людина",80),
 ...
clauses
...
isErroneous(місто_V),
...
end implement dynamicClass
```

На етапі компіляції Пролог виведе повідомлення про помилку, оскільки факт не ініціалізований при оголошенні.

Константи та домени можуть оголошуватися в деклараціях класів, інтерфейсів і в імплементації. За своєю сутністю вони, природно, мають статичний характер.

Оголошення, зроблені в імплементації, видимі тільки всередині цієї ж імплементації. Оголошення констант і доменів, зроблені в декларації класу або інтерфейсу, – є глобальними. Конкретне використання здійснюється шляхом додавання префікса декларації відповідного класу або інтерфейсу у форматі

<ім'я декларації класу або інтерфейсу> :: <ім'я домену або константи>.

Клаузи предикатів, оголошені статичними, не можуть звертатися до динамічних сутностей (предикатів і фактів), як наприклад:

```
implement dynamicClass
```

```
clauses
staticPred():-
...
dynamicPred(),
...
end implement dynamicClass
```

Клаузи предикатів, оголошених динамічними, такого обмеження не мають. Як статичні так і динамічні факти можуть зберігати покажчики на динамічні предикати і екземпляри класів.

9.5. Елементи класу

Коли говорять про об’єкти, мають на увазі, що вони є *екземплярами класів*. Це означає, що клас є свого роду формою (зразком), яка визначає, які дані й функції будуть включені в об’єкт цього класу. Тобто, клас можна представити як деякий шаблон, що визначає формат об’єкта. Клас є описом сукупності схожих між собою об’єктів. А об’єкти є втіленням властивостей, притаманних класу, до якого вони належать.

Клас може мати елементи, загальні для всіх об’єктів цього класу.

Наступний код програми дозволяє підраховувати кількість створених об’єктів класу *person*. Лічильник буде збільшуватися кожного разу, коли створюється новий об’єкт, і ніколи не буде зменшуватися. Слід розширити оголошення класу, додавши предикат *getCreatedCount* (отримати значення лічильника створення), який зможе повертати поточне значення лічильника:

```
class person: person

constructors
new: (string Name).

Predicates
getCreatedCount:() -> unsigned Count.


end class person
```

Видимий ззовні предикат класу, оголошений в оголошенні класу, тоді як видимий ззовні об’єктний предикат оголошений в інтерфейсі. З цього правила немає винятків – неможливо оголосити об’єктний предикат в оголошенні класу, і неможливо оголосити предикат класу в розділі інтерфейсу.

Після оголошення, кожен предикат повинен бути визначений в розділі *implement* класу. Те ж саме потрібно зробити для факту, який буде зберігати лічильник. Цей факт повинен бути фактом класу, тобто бути загальним для всіх об’єктів. У розділі *implement* класу можна оголосити елементи об’єкта як локальні (*private*), так і *елементи класу*. Для оголошення елементів класу потрібно вставити ключове слово *class* перед відповідним оголошенням. Наприклад, імплементація класу *person* може бути представлена таким чином:

```
implement person

class facts
createdCount: unsigned: = 0.

clauses
getCreatedCount ( ) = createdCount.

Facts
name: string.

clauses
new (Name): -name: = Name,createdCount: = createdCount +1.

clauses
getName () = name.

clauses
setName (Name): -name: = Name.


end implement person
```

Було додано факт класу *createdCount* і відбулася ініціалізація його нульовим значенням. Також додано клауз для предиката *getCreatedCount*, який повертає поточне значення змінної *createdCount*. І, нарешті, додано в текст клауз-конструктора рядок, який збільшує *createdCount* на одиницю.

У конструкторі два присвоювання мають схожий вигляд, але одне присвоювання *createdCount: = createdCount +1*, оперує з фактом класу, змінює стан класу, а інше – *name: = Name*, що оперує з фактом об’єкта, змінює стан об’єкта.

9.6. Модулі

Введення поняття класу є розвитком ідей модульності. У класі поєднуються структури даних і функції їхнього опрацювання.

Модулі – це особливі варіанти класів, які взагалі не породжують об’єктів, тому вони діють, як модулі, а не як класи. Клас, не конструює об’єкти (тобто, модуль), оголошується без вказівки об’єктного типу:

```
class io      % тип не вказано!
Predicates
write: (string ToWrite).
write: (unsigned ToWrite).
end class io
```

Такий клас не може породжувати об’єкти.

9.7. Конструктор і доступ

Конструктори використовуються для створення об’єктів та оголошуються явно у секціях *constructors* класів оголошення і реалізації. Коли об’єкт більше не потрібний, він повинен бути знищений деструктором для звільнення ресурсів, виділених на об’єкт.

Конструктор складається з двох зв’язаних предикатів: перший – це функція, яка повертає новий зконструйований об’єкт (виділяє пам’ять для збереження об’єктів); другий – предикат об’єкту, який використовується при ініціалізації успадкованих об’єктів. Як правило, це предикат-функція *new()*, але може бути й будь-який інший, оголошений як конструктор у розділі *constructors* декларації класу. Виконання предикату *delete()* в класі виконує явний деструктор для класу об’єктів. Предикати *new* і *delete* повинні бути процедурами.

Предикати *new* і *delete* мають багато рис звичайного члена-предикату, але вони мають і деякі особливості. Розміщуючи предикат *new*, можна викликати конструктори базових класів, використовуючи наступний синтаксис:

```
base_class_name :: new
```

У класі можуть бути оголошені кілька конструкторів і деструкторів з різним розміщенням або різними аргументами.

Перш ніж робити які-небудь посилання на об’єкт класу, об’єкт повинен бути створений за допомогою виклику будь-якого, оголошеного для класу, конструктора *new*. Це буде перевірено компілятором.

Після виклику деструктора *delete*, зазначений об’єкт буде знищений, і будь-які спроби використати його будуть неуспішними. Видалення об’єкту викличе автоматичне видалення всіх нестатичних фактів цього об’єкту і звільнення пам’яті, що була використана даним об’єктом.

Явно оголошений предикат *new* може бути викликаний різними способами:

- у конструкторі класу об’єктів

  Created_Object_Identifier = class_name :: new [(аргументи)]

  Наприклад:

  D = дитина :: new

  При виклику в якості конструктора, предикат *new* створює новий об’єкт (*екземпляр*) класу і повертає посилання на створений об’єкт.

- в якості члена предикату класу

  [Object_Identifier :] [class_name ::] new [(аргументи)]

  Наприклад:

  D: батько :: new

На імена класів посилаються за допомогою символу '::', наприклад, *дитина :: new(...)*. А об’єктні предикати посилаються на об’єкти з використанням символу ':', наприклад, *D: батько*.

Кожен клас має принаймні один конструктор. Якщо клас, який створює об’єкт, не оголошує ніяких конструкторів, то конструктор існує неявно (новий/0) в оголошенні класу.

Таким чином:

```
class aaa
end class aaa
```

одинаково, що

```
class aaa

constructors
new : ( ).
end class aaa
```

Якщо треба реалізувати неявний (не оголошений0 конструктор, то це записується наступним чином:

```
implement aaa
end implement aaa
```

одинаково, що

```
implement aaa

clauses
new ( ).


end implement aaa
```

враховуючи, що *aaa* має конструктор по замовчуванню.

З використанням коду, наведеного нижче, можна задати ціль, в якій створюється об’єкт, що використовується класом *іпб* для видачі на друк імені персони

```
goal
P = person :: new ("Іван"),
Імя = P: отримІмя (),
іпб :: write (Імя).
```

У першому рядку відбувається виклик конструктора (*constructor*) *new* класу *person*. Створений об’єкт запам’ятовується в змінній *P*. У другому рядку, змінної *І’мя* присвоюється результат виконання об’єктного предиката *отримІмя* об’єкта *P*. В останньому рядку відбувається виклик предикат *write* класу *іпб* з передачею йому імені *Імя*.

У наступному прикладі клас *bb_class* явно оголошує конструктор, який не є конструктором по замовчуванню. Надалі клас не має конструктора по замовчуванню

```
class bb_class aa

constructors
newFormFile : ( file File).


end class
```

Таким чином конструктори є функціями, які повертають об’єкти, навіть якщо ці конструктори не оголошені, як функції. Тип значення, що повертається,

визначається виходячи з оголошення класу, до якого застосовується конструктор.

Явне оголошення предикатів *delete* може бути викликане у два різні способи:

- у деструкції класу об’єктів

  Object_Identifier: [class_name ::] delete [(аргументи)]

  Наприклад:

  D: батьки :: delete,

- в якості члена предикату класу

  class_name :: delete [(аргументи)]

  Наприклад:

  батьки :: Delete (),

Видалення члена об’єкту класу не приводить до видалення всього об’єкту.

9.8. Інтерфейси як об’єктні типи

Інтерфейси є об’єктними типами. Інтерфейс має ім’я і визначає набір іменованих об’єктних предикатів. Інтерфейси структуровані в ієрархії засобів підтримки, структура яких подібна решітки, корінь якої в об’єкті інтерфейсу. Якщо об’єкт має тип, оголошений як інтерфейс, він також має тип будь-яких підтримуємих інтерфейсів. Таким чином, ієрархія засобів підтримки – це ієрархія типів. Інтерфейс являється підтипом усіх підтримуємих інтерфейсів. Отже, об’єкт підтримує інтерфейс. Якщо, наприклад, інтерфейс називається *X*, то це означає, що об’єкт – *X* або *X* об’єкт.

Можна використовувати інтерфейси скрізь, де використовуються звичайні необ’єктні типи. Наприклад, в оголошенні предиката:

```
class mail

predicates
sendMessage: (person Recipient, string Message)


end class mail
```

Предикат *mail :: sendMessage* має типи аргументів: *person* (тобто, ім’я інтерфейсу) і *string*.

9.9. Різноманітність імплементацій

Можна створити декілька різних класів, які можуть створювати однакові об’єкти *person*. Слід просто оголосити класи з розділами *implement*, які містять свої конструктори об’єктів *person*. Розділи *implement* можуть бути різними. Наприклад, можна створити клас, який зберігає персони в базі даних, а не в факт-змінній. Ось, оголошення такого класу:

```
class personInDB: person

constructors
new: (string DatabaseName, string Name).

end class personInDB
```

Наступний код демонструє, що об’єкти однакових типів можуть мати різні розділи *implement*:

```
implement personInDB

facts
db: myDatabase.
personID: unsigned.

clauses
new (DatabaseName, Name): -
db: = myDatabase :: getDB (DatabaseName),
personID: = db: storePerson (Name).

clauses
getName () = db: getPersonName (personID).

clauses
setName (Name): -
db: setPersonName (personID, Name).

end implement personInDB
```

Розрізняється не тільки внутрішня поведінка об’єкту, а й внутрішній стан також має різну структуру і зміст.

9.10. Категорований поліморфізм

Поняття поліморфізму може бути інтерпретовано як здатність класу належати більш ніж до одного типу.

Об’єкти одного типу можуть використовуватися в однакових контекстах, незалежно від відмінності в розділах *implement*. Можна, наприклад, послати

повідомлення якійсь персоні, використовуючи поштовий клас, оголошений заздалегідь, не звертаючи увагу, на те, яким конструктором об’єкт *person* був створений: *person* або *personInDB*:

```
goal
P1 = person :: new ("Іван"),mail :: sendMessage (P1, "Привіт Іван, ..."),
P2 = personInDB :: new ("Ігор"),mail :: sendMessage (P2, " Привіт Ігор, ...").
```

Ця поведінка відома, як категорована: в деяких контекстах, об’єкти, створені за допомогою одного класу, використовуються нарівні з об’єктами, створеними за допомогою іншого класу, якщо обидва об’єкти мають один і той же тип, який потрібно контекстом.

Предикат *mail :: sendMessage* приймає об’єкти будь-якого класу *person*, тому цей предикат є, в певному сенсі, поліморфним.

9.11. Supports – як розширення типів

Нехай програма має справу зі специфічними персонами, а саме, – з користувачами програм. Користувачі є персонами з іменами і, крім того, вони мають паролі. Треба створити новий інтерфейс (тип) об’єкта для користувачів, який би встановлював, що користувач є персоною з паролем. Для цього слід використати кваліфікацію *supports*:

```
interface user support person % підтримує person
predicates
змінаПаролю : (string Старий, string Новий, string Підтвердження) determ
перевіркаПаролю : (string Пароль) determ
end interface user
```

Інтерфейс *user* підтримує інтерфейс *person*. Це означає, що об’єкти типу *user* виконуватимуть також предикати, оголошені в інтерфейсі *person*, а саме, *отримІмя* і *привлІмя* та об’єкти типу *user* є одночасно об’єктами типу *person* і, тому, можуть використовуватися в тих же контекстах, що і об’єкти *person*.

Припустимо, що є деякий клас *user*:

```
class user : user

constructors
new : (string Імя, string Пароль).

end class user
```

Тоді об’єкти цього класу можуть бути використані предикатом *mail :: sendMessage*:

```
goal
P = user :: new ("Іван", "МійНовий Пароль"),mail :: sendMessage (P, "Привіт Іван, ...").
```

Підтримка інтерфесом інших інтерфейсів означає, що об’єкти даного типу повинні підтримувати (надавати право виклику і виконувати) предикати всіх інтерфейсів, які підтримуються і такі об’єкти є також об’єктами всіх типів, які відповідають підтримуваним інтерфейсам.

Кваліфікація *supports* (підтримка) породжує, таким чином, ієрархію підтипів – тобто *user* є підтипом *person*.

9.12. Object – первинний тип

Інтерфейс *object* неявно підтримує всі інтерфейси. Інтерфейс *object* є інтерфейсом, що не мають змісту, тобто в ньому немає оголошених предикатів. Об’єктний інтерфейс *object* підтримує, прямо або непрямо, будь-який інший інтерфейс, тому всі об’єкти мають тип об’єктного інтерфейсу *object*. Об’єктний інтерфейс *object* є супертипом для будь-якого об’єкту.

9.13. Спадкування

У Visual Prolog успадковування коду відбувається тільки у реалізації класу. Visual Prolog має множинну спадковість. Успадкувати від класу можна, вказавши у спеціальній секції спадкування *inherits*.

Для застосування класу *user*, можна використовувати властивості одного з наших класів *person*. Клас *user* схожий на клас *person*, за винятком того, що *user* має справу ще й з паролями. Треба, наприклад, щоб клас *user* успадковував розділ *implement* з класу *person*. Це можна зробити за допомогою кваліфікації *inherits (успадкування)*, як показано нижче:

```
implement user
inherits person

facts
пароль: string.

clauses
new (Імя, Пароль): -person :: new (Імя),пароль: = Пароль.

clauses
```

```
встановитиПароль(Старий, Новий, Підтвердження) :-
                        перевіркаПаролю(Старий),
                        Новий= Підтвердження,
                        пароль:=Новий.
clauses
перевіркаПаролю(Пароль):-
                                пароль = Пароль.

end implement user
```

Розділ *implement* підтверджує, що цей клас успадковує (*inherits*) властивості класу *person*. Звідси випливає, що об’єкт *person* вбудовується в кожен створений об’єкт user і всі предикати, оголошені в інтерфейсі *person*, можуть бути успадковані з класу *person* в клас *user*.

Коли успадковується який-небудь предикат, визначати його в розділі *implement* не потрібно, тому що успадковується визначення предиката з розділу *implement* успадкованого класу.

Можна отримати той же результат за допомогою наступного коду (клаузи для предикатів, паролі залишаються тими ж):

```
implement user

facts
person: person.
пароль: string.

clauses
new (Імя, Пароль): -person: = person :: new (Імя),
                    пароль: = Пароль.
clauses
отримІмя () = person: отримІмя().

clauses
привлІмя(Імя): - person: привлІмя(Імя).

end implement user
```

Дія цього коду схоже на дію попереднього фрагмента, але розмір тексту збільшився. У цьому фрагменті успадкування з класу *person* не відбувається. Замість цього створено об'єкт *person* і він збережений в однойменному факті-змінної. Замість спадкування коду *отримІмя* і *привлІмя* вони просто перевизначені і тепер звертаються за вирішенням до об’єкта з факт-змінної.

Такі звернення за рішенням можна скоротити за допомогою ключового слова *delegate*. Код у цьому випадку скоротиться.

```
implement user
delegate interface person to person

facts
person: person: = erroneous.
пароль: string.

Clauses
new (Імя, Пароль): -person: = person :: new (Імя),
пароль: = Пароль.

end implement user_class
```

Функціонування те ж, але виклики предикатів, що відносяться до об’єкта *person* транзитом передаються в нього і є можливість динамічно змінити значення в факт-змінної на інший об’єкт, наприклад, на об’єкт класу *personInDB* простим привласненням нового об’єкта фактом-змінної.

Visual Prolog обробляє такі непрямі звернення досить ефективно, але в разі спадкування обробка все ж відбувається набагато швидше; дає можливість багаторазового успадкування, тобто, є можливість одночасно успадковувати багато класів.

Кожен клас, крім абстрактних класів, повинен мати оголошення класу і класу реалізації, яка слідує за оголошенням класу, наприклад:

```
class XString          % оголошення класу
predicates
writeString          % оголошення предикату
endclass

implement XString  % виконати

facts
determ strDB(string)

clauses
writeString :- assert(strDB("Привіт Стратегія\n")), strDB(X),%Вставити факт
write(X), retractall(strDB(_)). % прочитати та видалити факт

endclass XString

goal
NewObject = XString::new,
```

```
NewObject:writeString,
NewObject:delete.
```

Результат виконання Пролог-програми:

```
Привіт Стратегія
NewObject=6820290
1 Solution
```

Абстрактний клас – це визначення класу без реалізації. Використовується для забезпечення загальних понять – визначення інтерфейсів. Метою абстрактного класу є оголошення деяких віртуальних предикатів. Абстрактний клас може використовуватися тільки в якості базового класу для інших класів, не може створювати об’єкти. Неможливо оголосити факти і статичні предикати в абстрактному класі. У разі, коли абстрактний клас успадковує деякі базові класи, вони повинні також бути оголошені абстрактними. Абстрактний клас визначається за ключовими словами *abstract*. Наприклад, *abs_cl* клас називається «*абстрактний клас*», тому що немає об’єктів *abs_cl* класу. Він існує виключно для виведення інших класів:

```
class abs_cl      % абстрактний клас

predicates
writeString1()
writeString2()

endclass abs_cl

class na_cl : abs_cl

predicates
writeString1()
writeString2()
endclass na_cl
implement na_cl

clauses
writeString1():-write( "1-й абстрактний клас.\n" ).
writeString2():-write( "2-й абстрактний клас.\n" ).
endclass na_cl

goal
Obj = na_cl::new(),
Obj:writeString1(),
Obj:writeString2(),
Obj:delete().
```

Результатом є:

```
1-й абстрактний клас.
2-й абстрактний клас.
Obj=6820270
1 Solution
```

Клас може бути похідним від іншого класу (класів), який називається базовим (батьківським) класом(класами). Похідний клас успадковує властивості, факти і предикати із зазначених базових класів.

Успадковані предикати або факти можуть бути перевизначені в похідному класі. Глобальні предикати також можуть бути перевизначені усередині класу. Члени базового класу, перевизначені в похідному класі, можуть бути доступні з похідного класу з явним відбірковим ім'ям члена базового класу. Синтаксис:

```
[object_identifier :] [base_class_name ::] member_name[(arguments)]
```

Приклад:

```
class person

predicates
procedure add_імя( string ) - (i)
procedure add_батько( person ) - (i)
procedure add_мати( person ) - (i)
nondeterm write_info()

endclass person

class службовець : person

predicates
procedure add_компанія(string Імя) -(i)
nondeterm write_info()
endclass службовець

implement person

facts
імя( string )
батько( person )
мати( person )

clauses
add_імя(Name):-assert(імя(Імя)).
add_батько(Obj_person):- assert(батько(Obj_person)).
add_мати(Obj_person):- assert(мати(Obj_person)).
write_info():-імя(X),write("Імя=",X),nl,fail.
write_info():-батько(F),write("Батько:\n"),F:person::write_info(),fail.
write_info():-мати(M),write("Мати:\n"),M:person::write_info(),fail.
```

```
write_info().
endclass person
implement службовець

facts
компанія(string Імя)

clauses
add_компанія(Імя):-assert(компанія(Імя)).
write_info():-this(O),O:person::write_info(),fail.
write_info():-компанія(X),write("Компанія=",X),nl,fail.
write_info().

endclass службовець

goal
F = person::new(),
F:add_імя("Ігор"),
O = службовець::new(),
O:add_імя("Назар"),
O:add_батько(F),
O:add_компанія("PDC"),
O:write_info(),
F:delete(),
O:delete().
```

Відповідь:

```
Імя=Назар
Батько:
Імя=Ігор
Компанія=PDC
F=68202A0, O=68231EC
1 Solution
```

9.14. Пакети

Базовий модуль організації коду у Visual Prolog – це пакет, який використовують для організації і структурування. Використання пакетів забезпечує однорідність у принципах структурування серед різних пакетів. Пакети визначають стандарт для інструментів структурування і простоту сумісного використання вхідного коду серед пакетів.

Пакети представляють собою деяке загальне ім’я для всіх інтерфейсів і класів. Кожна декларація або кожна реалізація кожного інтерфейса або класу з пакету розміщується у окремому файлі, які зберігаються у тій же директорії, що й пакет. Концепція пакетів використається джля об’єднання деяких зв’язаних інтерфейсів і класів.

Пакети можуть відігравати роль деяких бібліотек класів. Пакети можна використовувати в програмі замість прямого розміщення всіх використовуємих інтерфейсів і класів.

9.15. Видимість, приховування та кваліфікації

Визначення інтерфейсу, оголошення і реалізації класу являються областями видимості (не можуть бути вкладеними). Часто реалізація розширює області видимості для оголошення класу. Видимість є завжди в області. Це означає, незалежно від того, де в області що оголошено, все це видно у всій області видимості.

Загальнодоступні імена з підтримуємих інтерфейсів та *super-classes* доступні всередині області видимості (тобто без кваліфікації), якщо однозначно відомо, звідки вони взяті. Неможна використовувати ім’я з неоднозначним походженням. Всі нечіткості під час виклику предикату можуть бути видаленими, якщо кваліфікувати виклик предикату з іменем класу, наприклад, aa::p. Кваліфікація також використовується для виклику об’єкта предиката *super-classes* для поточного об’єкту.

Visual Prolog має наступні приховані ієрархії: локальні, суперкласу та відкриті області видимості. Ієрархія означає, що локальне оголошення буде приховувати декларацію суперкласу, але всі суперкласи мають однакову перевагу. У випадку, якщо два суперкласи містять суперечливі декларації, до них можна отримати доступ тільки безпосередньо через кваліфікацію.

Наприклад, існує інтерфейс *aa* та клас *aa_class*:

```
interface aa

predicates
p1 : () procedure().
p2 : () procedure().
p3 : () procedure().

end interface

class aa_class : aa
end class
```

Також представим клас *bb_class*:

```
class bb_class

predicates
p3 : () procedure().
p4 :() procedure().

end class bb_class
```

В контексті цих класів слід розглянути реалізацію класу *cc_class*

```
implement cc_class inherits aa_class
open bb_class

predicates
p2 : () procedure().
p5 : () procedure().

clauses
new() :-
p1(),          % aa_class::p1 невидимий
p2(),          % cc::p2 (приховує aa_class::p2)
aa_class::p2(),     % aa_class::p2 виден
p3(),          % неправильний виклик: aa_class::p3 or bb_class::p3
aa_class::p3(),  % aa_class::p3 видимий
bb_class::p3(),  % bb_class::p3
p4(),            % bb_class::p4
p5(),            % cc::p5
end implement cc_class
```

Запитання. Завдання

1. Яким критеріям повинна задовольняти об’єктно-орієнтована система?
2. Призначення інкапсуляції в об’єктно-орієнтованій системі?
3. Які основні елементи об’єктної моделі Visual Prolog?
4. Що таке об’єкт в об’єктній моделі Visual Prolog?
5. Чим характеризується стан об’єкта?
6. Які основні засоби використовуються при роботі з об’єктами?
7. Інтерфейси у Visual Prolog?
8. Які класи можуть бути використані як генератори об’єктів?
9. Що містить декларація класу у Visual Prolog?
10. Що представляє собою код, що міститься між ключовими словами *implement* і *end implement?*
11. Поясніть вираз «концепція області видимості»?

12. Яким чином декларуються і використовуються факти?
13. Якими є оголошення констант і доменів, зроблені в декларації класу або інтерфейсу?
14. Що таке модуль?
15. Що представляє собою конструктор?
16. Що таке поліморфізм?
17. Що можна сказати про об’єктний інтерфейс *object?*
18. Що таке *Абстрактний клас*?

Тест для самоконтролю знань з розділу 9[Ғ]

1. Як називається процес відокремлення один від одного окремих елементів об’єкта, що визначають його будову та поведінку?
   а) спадкування;
   б) інкапсуляція;
   в) індивідуальність.
2. Як визначається об’єкт?
   а) як властивість;
   б) як предмет, що прихований від зовнішнього середовища;
   в) як предмет або явище, що мають чітко визначену поведінку.
3. Властивості об’єкта, що відрізняє його від усіх інших об’єктів – це …
   а) спадкування;
   б) інкапсуляція;
   в) індивідуальність;
   г) всі відповіді вірні.
4. Спадкування – це …
   а) механізм ООП, який дозволяє побудову нового класу на основі існуючого, з можливістю додавання чи перевизначення даних і методів;
   б) процес відокремлення один від одного окремих елементів об’єкта, що визначають його будову та поведінку;
   в) властивості об’єкта, що відрізняє його від усіх інших об’єктів.
5. Що являється основним засобом при роботі з об’єктами, об’єктними типами і класами?
   а) *class declarations*;
   б) *implementations*;
   в) *interfaces*;
   г) всі відповіді вірні.
6. Що таке інтерфейси?
   а) *це* класифікація об’єктних типів;
   б) це властивості об’єкта, що відрізняє його від усіх інших об’єктів;

[Ғ] Усі запитання мають один правильний варіант відповіді.

в) це об’єктні типи, що описують доступ до об’єктів ззовні об’єкта.

7. Клас – це …?
   а) іменований набір оголошень предикатів;
   б) множина об’єктів, пов’язаних спільністю структури і поведінки;
   в) це об’єктні типи, що описують доступ до об’єктів ззовні об’єкта.
8. Декларацією класу є …
   а) іменований набір оголошень предикатів;
   б) множина об’єктів, пов’язаних спільністю структури і поведінки;
   в) сутність, невіддільна від імплементації класу з тим же ім’ям.
9. Визначення області видимості конкретного класу?
   а) у будь-якому місці після оголошення предикатів;
   б) у будь-якому місці після ключового слова *class*;
   в) лежить між ключовими словами *implement – end implement*.
10. Клас називається статичним, якщо …
   а) клас має інтерфейс;
   б) інтерфейс класу відсутній.
11. Оголошення констант і доменів є глобальними …
   а) якщо зроблені в декларації класу;
   б) якщо зроблені в декларації інтерфейсу;
   в) всі відповіді вірні.
12. Модулі – це …
   а) особливі варіанти класів, які взагалі не породжують об’єктів;
   б) особливі варіанти класів, які породжують об’єкти.
13. Абстрактний клас може використовуватися …
   а) тільки в якості базового класу для інших класів, може створювати об’єкти;
   б) тільки в якості базового класу для інших класів і не може створювати об’єкти.

РОЗДІЛ 10. ФАЙЛОВА СИСТЕМА VISUAL PROLOG

Visual prolog володіє гнучкою системою вводу-виводу і маніпулювання файлами.

Файл – це іменована сукупність даних, записаних на диску. Файл складається з компонентів – *елементів*. При читанні або запису файлова змінна переміщається до чергового компоненту і робить його доступним для обробки.

Файли користувача описуються у розділі опису доменів у відповідності з синтаксисом:

file = <символічне ім'я файла1>; ...; <символічне ім'я файлаN>

При описі файлових доменів тип домену *file* розташовується зліва від знака рівності, а праворуч розміщуються *символічні імена* файлів – внутрішні чи логічні імена (не слід плутати із зовнішніми або фізичними іменами файлів). Символічне ім’я файлу повинне розпочинатися з рядкової (малої) літери.

Крім користувальницьких файлів, є стандартні файли (або потоки), які не потрібно описувати в розділі опису доменів, такі як:

- *stdin* (стандартний поток введення);
- *stdout* (стандартний поток виводу);
- *stderr* (стандартний поток виведення повідомлень про помилки);
- *keyboard* (клавіатура);
- *screen* (монітор).

Наприклад:

```
domains
file = input                    % символічне ім’я файлу

goal
% відкриває файл для читання
openread(input,"dd.txt"),
% переопреділяє поточний пристрій зчитування
readdevice(input),
% читає текстовий рядок
readln(L1),write(L1),nl,
readln(L2),write(L2),nl,
readdevice(keyboard),
write("введіть текст: "),
readln(L3),
write(L3),nl.
```

По замовчуванню стандартним пристроєм для потоку вводу є клавіатура, а стандартним пристроєм для потоку виводу – монітор. Щоб розпочати роботу з файлом користувача, його треба відкрити, а по завершенню роботи – закрити. Стандартні потоки введення/виведення відкривати і закривати не треба.

```
domains
file = myfile

goal
openwrite(myfile,"dd.txt"),
writedevice(myfile),
write("Файл з одним рядком\n"),
writedevice(screen),
write("\n Вивести на екран"),
writedevice(myfile),
closefile(myfile),
writedevice(Dev),
write("\n Зверніть увагу на автозбереження"," встановлене: ",Dev),
write("\n при закритті файлу").
```

Файлова система мови Visual Prolog включає стандартні предикати відкриття і закриття файлів, читання з файлу і запису у файл, зміни даних у файлі, а також дозапису в існуючий файл. Тим самим забезпечуються чудові можливості ефективної обробки файлів.

Зазвичай клавіатура є пристроєм введення інформації, а поточним пристроєм виведення – є екран монітору. Однак, можна визначити й інші пристрої введення або виведення в процесі виконання програми. Наприклад, введення з файлу, а виведення – на пристрій друку.

Для всіх пристроїв введення/виведення виконується ідентично. Спочатку файл повинен бути відкритий і зробити це можна одним із способів:

- *для читання;*
- *запису;*
- *додавання;*
- *модифікації.*

Файл, відкритий для будь-якої дії, крім читання, запису, додавання і модифікації. Файл, відкритий для будь-якої дії, крім читання, необхідно закрити після закінчення роботи. В іншому випадку інформацію можна втратити.

Одночасно можуть бути відкриті кілька файлів.

Відкриття та закриття файлів при кожному зверненні займає значно більше часу, ніж звернення до відкритих файлів з метою запису або читання інформації з них.

При відкритті файлу здійснюється зв'язування символічного імені з *дійсним ім'ям*, прийнятим у відповідній операційній системі. Це символічне ім'я і використовується для направлення вводу/виводу.

Символічне ім'я файлу (воно ще називається логічним ім'ям) повинно починатися з рядкової латинської літери і оголошується в описі домену *file*. Цей опис має бути єдиним у програмі, наприклад:

```
file = mybase1
```

або

```
file = database1; dfile2; dfile3,
```

тобто у цьому випадку оголошено три логічних імені для ототожнення їх з реальними файлами DOS.

10.1. Визначення виду доступу до файлу

Доступ до файлу може виконуватися у двох «модах» – бінарній та текстовій.

Предикат *filemode (i,i), (i,o)*

Для визначення виду доступу використовується спеціальний предикат

```
filemode (file SymbolicFileName, integer ModeOfFile)
```

Параметр *ModeOfFile* приймає одне з двох значень:

```
ModeOfFile = 0 (Двійковий режим),
ModeOfFile = 1 (Текстовий режим).
```

В текстовому режимі під час запису до нових рядків додаються символи «*повернення каретки*»/«*переведення рядка*», а при читанні ці символи інтерпретуються як новий рядок. У двійковому режимі ніяких перетворень не виконується (для читання можна використати тільки предикат *reafchar*).

Для того, щоб працювати з файлом на зовнішньому носії інформації, його треба відкрити або створити. Відкрити файл можна для читання, запису та модифікації. Перед тим, як це зробити, у файлі *<ім'я_проекта>.inc* в разділі

global domains слід створити символічне ім'я файлів, розділяючи їх крапкою з комою відповідно синтаксису оголошення файлів. Наприклад, є оголошення:

```
global domains

db_selector = browselist_db
file = fileselector1; fileselector2
```

Слід додати нові імена:

```
global domains

db_selector = browselist_db
file = fileselector1; fileselector2; filein; filein2; fileout
```

10.2. Відкриття файлів

Відкрити можна практично необмежену кількість файлів – стільки, скільки дозволяють установки операційної системи. Предикати *openread*, *openwrite*, *openappend* та *openmodify* мають два вхідних параметра. Перший параметр – це внутрішнє символічне ім'я, вказане у розділі опису доменів, другий параметр – це рядок, який являє собою зовнішнє ім'я файлу.

Предикат *openread (i,i)*

Призначений для відкриття файлу тільки для читання.

Формат:

```
openread (file SymbolicFileName, string OSFileName)
```

Приклад:

```
domains
file = input   % оголошення домену

predicates
repfile(file)

clauses
repfile(_).
repfile(F) :-not(eof(F)),
repfile(F).

goal
% відкриття файлу тільки для читання
openread(input,"dd.txt"),
readdevice(input),
repfile(input),
readln(L),
write(L),nl,
fail.
```

Якщо файл з вказаним зовнішнім іменем не буде знайдено, предикат терпить невдачу та виводить відповідне повідомлення про помилку.

Предикат *openwrite (i,i)*

Відкриває файл тільки для запису. Цей предикат створює на диску новий файл. Якщо файл з вказаним зовнішнім ім'ям вже існує, він буде стертий. Якщо з якоїсь причини файл не може бути створено, предикат терпить невдачу та виводить відповідне повідомлення про помилку.

Формат:

```
openwrite (file SymbolicFileName, string OSFileName)
```

Приклад:

```
domains
file = myfile

goal
% Відкриває файл тільки для запису
openwrite(myfile, "dd.txt"),
writedevice(myfile),
write("line 1\n"),
write("line 2\n"),
write("line 3\n"),
closefile(myfile),
file_str("dd.txt", Str1),
display(Str1).
```

Предикат *openappend (i,i)*

Відкриває файл тільки для дозапису, який виконується в кінець файлу. Якщо файл з вказаним ім'ям не буде знайдено, предикат виводить відповідне повідомлення про помилку.

Формат:

```
openappend (file SymbolicFileName, string OSFileName)
```

Приклад:

```
domains
file = myfile

goal
file_str("dd.txt", "Створити файл з одного рядка\n"),
% Відкриває файл тільки для дозапису
openappend(myfile, "dd.txt"),
writedevice(myfile),
write("це другий рядок у файлі\n"),
closefile(myfile),
```

```
file_str("dd.txt", Str1),
display(Str1).
```

Предикат *openmodify (i,i)*

Відкриває файл для читання та запису одночасно. Якщо файл з вказаним ім'ям не буде знайдено, предикат виводить відповідне повідомлення про помилку.

Формат:

```
openmodify (file SymbolicFileName, string OSFileName)
```

Приклад:

```
domains
file = myfile

goal
file_str("dd.txt","Створити файл з двох рядків\n"),
% Відкриває файл для читання
% та запису одночасно
openmodify(myfile, "dd.txt"),
readdevice(myfile),
readln(L),
filepos(myfile,FilePos,0),
writedevice(myfile),
% Переміщає покажчик файлу в кінець
filepos(myfile,FilePos,0),
write("Перезапише другий рядок\n"),
closefile(myfile),
file_str("dd.txt", Str1),
display(Str1).
```

Предикат *existfile (i)*

Для того щоб провірити, чи існує файл з вказаним ім'ям і вказаному місті (шлях), використовується предикат *existfile* за форматом:

```
existfile (string OSFileName)
```

Цей предикат має один аргумент. Предикат є істина, якщо файл з ім'ям, вказаним у якості його єдиного параметра, існує, та хибний в іншому випадку:

```
goal: existfile("dd.txt")
/*Yes*/

goal: deletefile("dd.txt")
/*Yes*/

goal: existfile("dd.txt")
/*No*/
```

Важливо звернути увагу на те, що ці предикати зв’язують символічне ім’я файлу з фізичним ім’ям файлу, що відкривається. Тому, на відміну від інших мов програмування, у Пролозі немає необхідності перед операцією відкриття файлу проводити операцію зв’язування внутрішнього і зовнішнього імен файлу.

Оскільки символ "\", що зазвичай використовується для розділу імен каталогів, застосовується у Пролозі для запису кодів символів, вимагається використовувати замість одного зворотнього «слэша» два ("\\").

Наприклад, для того щоб вказати шлях

"D:\VP7.4\Doc\Visual Prolog Exemples"

треба записати рядок

"D:\\VIP7.4\\Doc\\Visual Prolog Exemples".

10.3. Перевизначення файлів

Для переходу від одного відкритого файлу до другого (перенаправлення потоків введення/виведення) призначені предикати *readdevice* и *writedevice*.

Предикат *readdevice (i), (o)*

Перевизначає поточний пристрій читання або повертає його ім’я.

Формат:

```
readdevice (file SymbolicFileName)
```

Приклад:

```
domains
file = input

goal
openread(input,"dd.txt"),
% Перевизначає поточний пристрій
readdevice(input),
readln(L1),write(L1),nl,
readln(L2),write(L2),nl,
% Перевизначає поточний пристрій
readdevice(keyboard),
write("Enter text: "),
readln(L3),write(L3),nl.
```

Наступний приклад:

```
openread(filein,"text.txt")"
openread(filein2,"text2.txt")"
openwrite(fileout,"forwrite.txt")
```

```
readdevice(filein),
readln(Str1),
write(Str1)
.....................
readdevice(filein2),
readln(Str),
write(Str),
.....................
closefile(filein),
closefile(filein2),
closefile(fileout),
```

Предикат *writedevice (i), (o)*

Призначає/дозволяє отримати ім’я поточного пристрою запису.

Формат:

```
writedevice (file SymbolicFileName)
```

Приклад:

```
domains
  file = myfile

goal
  openwrite(myfile,"dd.txt"),
  writedevice(myfile),
  write("Файл з одним рядком\n"),
  % screen - екран
  writedevice(screen),
  write("\n Виведення на екран"),
  writedevice(myfile),
  closefile(myfile),
  writedevice(Dev),
  write("\n Автозапис пристроєм", " встановленим в: ",Dev),
  write("\n під час закриття файлу ").
```

10.4. Виведення на друк

Предикат *write*

Універсальний оператор запису. Виводить об’єкти на друк за форматом:

```
write(e_1, e_2, e_3, ... , e_N)
```

з параметрами *(i, i, i, ..., i)*

Наприклад:

```
goal
Write("предикат", "може мати", " множину аргументів").
```

Використовується прологом для друку складних структур даних та списків.

Предикат *writef*

Має формат з аргументами (кількість не обмежена) з параметрами *(i, i, i, ..., i)*:

writef(string FormatString, Arg1, Arg2, Arg3, ...)

та дозволяє виводити на екран відформатований результат, використовуючи для цього форматні специфікатори (необов'язкові) формату *%-m.pf*:

- *дефіс* – вирівнювання по лівому краю;
- поле *m* – десяткове число, яке визначає мінімальну довжину поля;
- поле *p* – десяткове число, яке описує максимальне число символів у рядку;
- поле *f* – описує формати, відмінні від прийнятих за замовчуванням.

Visual Prolog розпізнає наступні специфікатори поля *f*:

- *f* – дійсні з фіксованою крапкою;
- *e* – дійсні в експоненціальному форматі;
- *g* – дійсні у форматі *f* або *e*;
- *d* – цілі як знакові десяткові числа;
- *D* – цілі як знакові довгі десяткові числа;
- *u* – цілі як беззнакові десяткові числа;
- *U* – цілі як беззнакові довгі десяткові числа;
- *x* – цілі як шістнадцятирічні числа;
- *X* – цілі як шістнадцятирічні довгі числа;
- *o* – цілі як восьмирічні числа;
- *O* – цілі як восьмирічні довгі числа;
- *c* – цілі як символи *char*;
- *B* – як бінарні (*binary type*);
- *R* – як числа посилань у зовнішніх базах даних;
- *P* – як предикатні значення;
- *S* – як символьні рядки.

Наприклад:

```
goal
writef("\n Демонстрація роботи предиката writef \n"),
A = "one",
B = 330.12,
C = 4.3333375,
D = "one two three",
writef("A = '%-7' \nB = '%8.1e'\n",A,B),
writef("A = '%' \nB = '%8.4e'\n",A,B),
writef("C = '%-7.7g' \nD = '%7.7'\n",C,D),
writef("C = '%-7.0f' \nD = '%0'\n",C,D),
writef("char: %c, decimal: %d, hex: %x, unsigned: %u",97,'a',33,-1).
```

Результат:

```
A = 'один'
B = ' 3.3E+02'
A = 'один'
B = '3.3012E+02'
C = '4.3333375'
D = 'один два'
C = '4 '
D = 'один два три'
Char : a, decimal: 97,"," hex: 21, unsigned: 65535
```

10.5. Читання та запис файлів

Предикат *file_str (i, o), (i, i)*

Читає всі символи файлу OSFileName в рядок StringVariable або, навпаки, записує вміст рядка в файл, в залежності від того, чи вільний другий параметр цього предиката. Формат:

```
file_str (string OSFileName, string StringVariable)
```

Першим вхідним параметром цього предиката є символічне ім'я файлу, а другим – рядок, в який зчитується вміст файлу або з якого записується інформація в нього.

Приклад:

```
predicates
extend(string,string)
getfilename(string,string)

clauses
extend(S,S) :- concat(_,".pro",S),!.
extend(S,S1) :- concat(S,".pro",S1).
getfilename("",Fname) :- dir("","*.pro",Fname),!.
getfilename(X,X1) :- extend(X,X1).

goal
comline(X),
```

```
getfilename(X,X1),
file_str(X1,S),
textmode(Rows,Cols),
makewindow(1,23,0,"EDITOR",0,0,Rows,Cols),
edit(S,S1,"","","",0,"",Ret),
removewindow,
Ret><1,!,
clearwindow,
write("\ [Enter] для запису filename: "),
readln(NewName),
file_str(NewName,S1).
```

Предикат *readchar (o)*

Читає один символ з пристрою введення (файл або клавіатура). Формат:

```
readchar (char CharVariable)
```

Наприклад:

```
counter(0,99,I),      % (i,i,o); лічильник від  1 to I;
write("Line %", I),nl,
readchar(_),
fail.
```

Предикат *readreal (o)*

Читає один символ з пристою введення (файл або клавіатура). Формат:

```
readreal(real RealVariable)
```

Приклад правильного запису: 127; -127.457E-5; 0.72; .99.

Наприклад:

```
goal
readreal(X).
/*123 % з клавіатури
X=123
1 Solution*/

goal
readreal(X).
/*-123.456E-5
X=-0.00123456
1 Solution*/

goal
/*readreal(X).
56
X=0.56
1 Solution*/

goal
readreal(X).
/*No Solution*/
```

```
goal
readreal(X).
/*aaaa
No Solution*/
```

Предикат *readint (o)*

Читає ціле число в межах (от -2147483648 до >2147483647 для 32 bit платформи). Формат:

```
readint(integer IntegerVariable)
```

Помилка виникає у випадку, коли введення не може бути перетворене в ціле.

Приклад:

```
goal
readint(X).
/*111
X=111
1 Solution*/

goal
readint(X).
/* -123
X=-123
1 Solution*/

goal
readint(X).
/* 999999
No Solution*/

goal
readint(X)
/* aaa
No Solution*/
```

Предикат *flush (i)*

Коли треба працювати з файлом в режимі читання/запису, слід після кожного запису використовувати предикат *flush*. Предикат записує вміст внутрішнього буфера в поіменований файл. Формат предиката має вигляд:

```
flush (file SymbolicFileName)
```

та визиває примусове очищення буфера.

```
goal
writedevice(printer),
```

```
write("Привіт"),
flush(printer),
writedevice(screen),
write("\n Виконати повернення каретки"),
readln(_).
```

Зазвичай він використовується при роботі з принтером. Корисний також під час відладки програми, коли треба писати деяку інформацію для трасування. (Flush значно уповільнює виконання програми.).

Предикат *filepos (i,i,i), (i,o,i)*

Вміст файлу можна розглядати як потік компонентів. Кожен компонент файлу знаходиться на якійсь позиції. Для того щоб дізнатися поточну позицію читання або запису у файлі, або для того, щоб змінити цю позицію, служить предикат *filepos* – використовується для реалізації нелінійного («гіпертекстового») доступу до файлу за форматом

filepos(SymbolicFileName,FilePosition,Mode), шаблон *(i,i,i), (i,o,i)*

Має три аргументи. Перший – це символічне ім'я файлу, другий – позиція всередині першого аргументу, яку потрібно дізнатися або встановити. Третій параметр *Mode* визначає «точку відліку» позиції:

- *0* - від початку файлу;
- *1* - відносно поточній позиції;
- *2* - відносно кінця файлу.

Предикат може бути використаний двояко. Якщо всі три його аргументу пов'язані, то позиція, з якої здійснюється читання або в яку проводиться запис, буде змінена у відповідності з числом, яким означений другий аргумент. Якщо його другий аргумент вільний, а перший і третій пов'язані, то другий аргумент буде означений поточною позицією читання або запису.

```
domains
file = my_file

predicates
repeat
position

clauses
repeat().
repeat():- repeat().
```

```
position():-
shiftwindow(2),
filepos(my_file, P1, 0),
writef(" вихідне значення: %", P1), nl,
shiftwindow(1).

goal
makewindow(2, 23, 23, " розміщення файлу ", 0, 40, 20, 40),
makewindow(1, 23, 23, " вихідний файл ", 0, 0, 20, 40),
dir("", "*.pro", Filename),
clearwindow(),
openread(my_file, Filename),
readdevice(my_file),
repeat,
readchar(Str),
position(),
write(Str),
eof(my_file).
```

Предикат *eof (i)*

Предикат призначений для контролю кінця файлу: успішний, якщо досягнуто кінця файлу, в іншому випадку він неуспішний. Вміст файлу можна розглядати як потік компонентів. Кожен компонент файлу знаходиться на якійсь позиції.

```
eof (file SymbolicFileName)
```

Предикат зазвичай використовується при організації рекурсивного зчитування всіх компонентів файлу.

Наприклад:

```
domains
nondeterm repeat
читати_і_обробити_рядки
обробити_рядок(symbol)
.....................
clauses
repeat.
repeat:-repeat.

читати_і_обробити_рядки:-
repeat,
readln(f,S),
обробити_рядок (S),
eof(f),
!,
closefile(f);
!.
```

10.6. Закриття файлів

Предикат *closefile (i)*

Призначений для закриття файлу. Завжди успішний, якщо вказаний файл існує. Формат:

```
closefile(file SymbolicFileName)
```

Приклад:

```
domains
file = myfile

goal
openwrite(myfile,"dd.txt"),
writedevice(myfile),
write(" Файл з одного рядка \n"),
closefile(myfile).
```

10.7. Операції з файлами

Для роботи з іменами файлів використовуються наступні предикати.

Предикат *filenameext (i,o,o),(o,i,i)*

Формат: *filenameext(string Name, string MainPart, string Extension)*

У випадку ***(i,o,o)*** – повертає власне ім’я файлу і розширення.

```
filenameext("myprog.vpr",Name,Ext),
write("Name=",Name, " Ext=",Ext),nl,

/*Результат:
Name=myprog Ext=.vpr*/
```

У випадку шаблону ***(o,i,i)*** – за відомим власним ім’ям файлу і розширення повертає ім’я файлу.

```
filenameext(FullName,"prolog",".err"),
write(FullName),nl,
filenameext(FullName1,"prolog.err",".exe"),
write(FullName1),nl.

/*Результат::
prolog.err
prolog.exe*/
```

Предикат *filenamepath (i,o,o), (o,i,i)*

Повертає повний шлях до файлу: шлях та ім’я файлу. Формат:

```
filenamepath(string QualName, string Path, string Name)
```

Так, наприклад, коли повний шлях до файлу є вхідним параметром (i,o,o) і задається з ім'ям диску від кореневого каталогу

```
QualNam ="d:\\VIP\\include\\IOdecl.CON",
```

то шлях до файлу буде:

```
Path="d:\\VIP\\include\\"
```

а ім'я файлу:

```
Name="IOdecl.con"
```

У другому випадку, коли шлях до файлу вказано від батьківського каталогу

```
QualName ="include\\iodecl.con",
```

предикат повертає:

```
Path="include\\"
Name="iodecl.con"
```

Коли шлях вказано від поточного каталогу:

```
QualName ="iodecl.con",
```

результатом є:

```
Path=""
Name="iodecl.con"
```

Приклад роботи предикату *filenamepath* з шаблоном параметрів *(i,o,o)*:

```
filenamepath("C:\\mycatalog\\myfile.txt",Path,Name),
write("Path=",Path," Name=",Name),nl,
```

Результат:

```
Path=C:\mycatalog\ Name=myfile.txt
```

Приклад роботи предикату *filenamepath* з шаблоном параметрів *(o,i,i)*

```
filenamepath(FullName,"C:\\mycatalog\\","anfile.txt"),
write(FullName),nl,
```

Результат:

```
C:\mycatalog\anfile.txt
```

Предикат *searchfile (i,i,o)*

Виконує пошук файлу серед списку вказаних шляхів і повертає повне ім'я файлу. Формат:

```
searchfile (string SearchPathList, string FileName, string FoundName)
```

Завершується неуспішно, якщо файл відсутній.

Наприклад:

```
goal
searchfile(".;..;C:\\","autoexec.bat",FoundName),
```

Якщо файл autoexec.bat збережений у кореневому каталозі диску C:\, то результатом буде

```
c:\autoexec.bat
```

Якщо виконати предикатом пошук всіх файлів за шаблоном *.bat

```
searchfile(".;..;C:\\","*.bat",FoundName),
```

результатом буде *C:*.BAT.*

Предикат *deletefile (i)*

Предикат видаляє файл, вказаний в якості його єдиного параметра, та має наступний формат:

```
deletefile (string OSFileName)
```

Предикат не може містити підставні символи. Виводить повідомлення у випадку, коли не може виконати дію.

Наприклад:

```
goal
deletefile("МійФайл.txt")


/*Yes*/
```

Предикат *renamefile (i,i)*

Перейменовує файл з новим ім’ям. Формат:

```
renamefile (string OldFileName, string NewFileName)
```

Буває неуспішним у випадку, коли файл не знайдено, або задане некоректне ім’я файлу.

```
goal
file_str("dd.txt", "Мій текст"),
renamefile("dd.txt", "МійФайл.txt"),
file_str("МійФайл.txt", X),
deletefile("МійФайл.txt"),
write(X),nl.
```

Предикат *copefile (i,i)*

Виконує копіювання файлу за форматом:

```
copyfile (string SourceFileName, string DestFileName)
```

Шлях до файлу може вказуватись включаючи диск, від кореневого каталогу, батьківського або поточного. Підставні символи не допускаються.

Наприклад:

```
goal
copyfile ("D:\\user||file1.pro", "\\file1.pro").
```

Предикат *disk (i), (o)*

Призначений для зміни поточного диску. Формат:

```
disk (string Path)
```

Якщо в якості аргументу використовується вільна змінна, то предикат повертає поточний каталог.

Приклад:

```
/* виконати */

goal
disk(CurPath),
write("РобочийКаталогДляСистемногоФайлу = ",CurPath),nl,
disk("c:\\"),
disk(CurPath1),
write("ТимчасовийРобочийКаталог = ", CurPath1),
nl,
upper_lower(UPCurPath,CurPath),
disk(UpCurPath),
disk(CurPath2),
write("РобочийКаталогWasSetIn_UPPER_CASE = ",CurPath2),
nl,
upper_lower(CurPath,LowCurPath),
disk("c:\\"),
disk(LowCurPath),
disk(CurPath3),
write("РобочийКаталогWasSetIn_lower_case = ",CurPath3),nl,
write("Буква завжди повертається у верхньому регістрі!"),nl,
readchar(_).
```

Практичне застосування описаних вище предикатів

У прикладі 1 демонструється виведення інформації з статичної бази у файл на диску і на екран монітора:

```
domains
file = laba

predicates
write_lines
data(symbol)
```

```
goal
openwrite(laba, "laba.dat"),
write_lines,
closefile(laba).

clauses
data("Запис1").
data("Запис2").
data("Запис3").

write_lines:-
        data(Line),
        write("",Line),nl,
        writedevise(laba),
        write("",Line),nl,
        writedevise(screen),
        fail.
write_lines.
```

У прикладі 2 вимагається створити предикат, що зчитує з клавіатури символи і поміщає їх у файл. При цьому символи, що набираються, на екран не виводяться.

```
domains
file = myfile
predicates
sozdf

clauses
sozdf:-readchar (X),
X <> '#',!,
write (X),      % write виконує запис символу в файл
sozdf.
sozdf.

goal
write ("Введіть символи для запису у файл, закінчення введення - #"), nl,
openwrite (myfile, "c: \ \ file1.dat"),
writedevice (myfile),
sozdf,
closefile (myfile),
writedevice (screen),
write ("Був створений файл C: \ file1.dat"),nl.
```

Щоб символи відображалися, необхідно після введення символу перенаправити пристрій введення на екран, відобразити символ, а потім знову перенаправити введення у файл. Для введення рядків таке доповнення не потрібно, тому що для зчитування рядків використовується вбудований

предикат *readln*, який, зчитуючи рядок в змінну, відображає її на екрані, наприклад,

```
domains
file = myfile
sozdf
run

clauses
sozdf:-readln (X),
!,
write (X), nl,   % тут write виконує запис рядки в файл
                 % і запис символу повернення каретки (nl)
sozdf.
sozdf.

run:-write ("Введіть рядки для запису у файл, закінчення – [Esc]"), nl,
openwrite (myfile, "c: \ \ file2.dat"),
writedevice (myfile),
sozdf,
closefile (myfile),
writedevice (screen),
write ("Був створений файл C: \ file2.dat"), nl.

goal
run.
```

Запитання. Завдання

1. Як оголошуються файли користувача?
2. Які файли у Пролог не потрібно оголошувати?
3. Які предикати належать до стандартних предикатів файлової системи Visual Prolog?
4. Що означає символічне ім’я файлу?
5. Як виконується доступ до файлу?
6. Порядок відкриття файлу з зовнішнього носія інформації.
7. Яким чином вивести на екран числові дані у певному числовому форматі, що встановлює кількість знаків після коми?
8. У яких випадках використовується предикат *Flush*?
9. Для чого зазвичай використовується предикат *eof* ?
10. Предикат, який повертає повний шлях до файлу?

Тест для самоконтролю знань з розділу 10[Ғ]

[Ғ] Усі запитання мають один правильний варіант відповіді.

1. Які базові стандартні предикати передбачено у Visual Prolog для запису?
   а) writef;
   б) write;
   в) nl;
   г) всі відповіді вірні.
2. Призначення предиката writef?
   а) для запису простого висновку;
   б)для запису форматованого висновку відповідно до покажчиків формату;
   в) для генерації переходу на новий рядок;
   г) всі відповіді вірні.
3. Призначення предиката readdevice?
   а) відкриття файлу для читання;
   б) переведення файлу у текстовий або двійковий режим;
   в) перепризначення поточного пристрою читання або отримання його імені;
   г) перепризначення поточного пристрою запису або отримання його імені.
4. Які предикати належать до спеціальних предикатів читання?
   а) openmodify, openwrite;
   б) readint, readreal, readchar, file_str;
   в) flush, filepos;
   г) renamefile, existfile.
5. Який з предикатів виконує перевірку потрапляння на кінець файлу при читанні?
   а) openwrite;
   б) readchar;
   в) filepos;
   г) iof.
6. Який з предикатів виконує управління позиціонуванням читання-запису?
   а) openwrite;
   б) readchar;
   в) filepos;
   г) iof.
7. Який формат має предикат writef?
   а) -m.pf
   б) %-m.pf
   в) %+m.pf
   г) %-f
8. Предикат readreal …
   а) читає ціле число в межах (от -2147483648 до >2147483647 для 32 bit платформи);
   б) читає один символ з пристою введення (файл або клавіатура);

в) записує вміст внутрішнього буфера в поіменований файл;

г) визначає поточну позицію читання або запису у файлі.

9. Предикат filenamepath …

а) повертає ім’я файлу;

б) виконує пошук файлу серед списку вказаних шляхів і повертає повне ім’я файлу;

в) повертає повний шлях до файлу;

г) виводить повідомлення у випадку, коли не може виконати дію.

10. Формат оголошення файлу користувача …

а) <символічне ім'я файла1>; ...; <символічне ім'я файлаN>= file

б) file = <символічне ім'я файла1>; ...; <символічне ім'я файлаN>

в) <символічне ім'я файла1>, ..., <символічне ім'я файлаN>= file

РОЗДІЛ 11. БАЗИ ДАНИХ У VISUAL PROLOG

Система внутрішніх баз даних (внутрішніх баз фактів) Visual Prolog є зручною і простою у використанні. Оскільки Visual Prolog представляє реляційну базу даних як колекцію фактів, його можна використовувати в якості мови запитів до внутрішніх баз фактів. Алгоритм уніфікації Visual Prolog автоматично вибирає факти з правильними значеннями для відомих аргументів та привласнює значення невідомим аргументам, доки його алгоритм пошуку з поверненням виводить всі рішення для заданого запиту.

Однак, потреби баз даних в оперативній пам'яті можуть дуже швидко перевищити можливості комп'ютера. Ця проблема може бути вирішена шляхом використання зовнішніх баз даних. За допомогою зовнішніх баз даних можна реалізувати різні види прикладних систем, забезпечуючи при цьому цілісність даних у разі їх оновлення, навіть у разі збою живлення.

11.1. Внутрішня база фактів (ВБФ)

11.1.1. Оголошення ВБФ

Внутрішня база фактів складається з фактів, які можна безпосередньо додавати чи видаляти з Пролог-програми під час її виконання.

Предикати, що описують внутрішню базу фактів, можна оголошувати у розділі програми *facts*, який розпочинається з ключового слова *facts* (синонім застарілого *database*). Він містить послідовності оголошень предикатів, які описують внутрішню базу фактів. Спосіб використання предикатів, оголошених таким чином, такий же як і предикатів, оголошених у розділі *predicates*.

У наступному прикладі предикат «людина» може використовуватися аналогічно предикатам «чоловік», «жінка», «дитина» з тією різницею, що стосовно предиката «людина» у процесі виконання програми можна виконувати операції вставки і видалення фактів:

```
domains

ім'я, адреса = string
вік = integer
стать = чоловік; жінка
```

```
facts
людина(імя, адреса, вік, стать)

predicates
чоловік(імя, адреса, вік)
жінка(імя, адреса, вік)
дитина(імя, вік, стать)

clauses
чоловік(Імя, яадреса, Вік):-
людина(Імя, яадреса, Вік, чоловік).
```

При поповненні внутрішньої бази фактів, вирази додаються тільки у вигляді фактів, а не правил. Причому факти внутрішньої бази фактів не можуть містити вільних змінних.

У Пролог-програмах допускається декілька розділів *facts*, з привласненням їм імен, наприклад:

```
facts — новаБД
відношення1(integer)
відношення2(real,string)
відношення3(string)
……..
goal
consult("exemple.dba", новаБД),retract(відношення1(1)),
   asserta(відношення1(0)),
   save(("exemple.dba", новаБД).
```

За замовчуванням внутрішній базі фактів привласнюється стандартне ім’я dbasedom.

Якщо Пролог-програма складається з одного модуля, що не оголошується як частина проекту, то вона може містити локальні непоіменовані розділи фактів. Візуальне середовище розробки компілює програмний файл як єдиний модуль тільки при використанні утіліти Test Goal, в іншому випадку непоіменований розділ фактів оголошується глобальним – *global facts*.

У модулі – вихідному файлі – імена предикатів внутрішньої бази фактів повинні бути унікальними. Не можна застосовувати однакові імена предикатів у двох різних розділах *facts* та у розділах *facts* і *predicates*.

Імена предикатів, визначених у локальних facts-розділах, є локальними для модуля, де вони оголошені, і не конфліктують з локальними іменами предикатів чи фактів, оголошених в інших модулях.

У оголошенні розділу *facts* можна використовувати ключові слова:

facts [— <databasename>]
[nocopy] [{ nondeterm | determ | single }]
dbPredicate ['(' [Domain [ArgumentName]] ')']*

Необов'язкові *nondeterm, determ ma single* оголошують режим детермінізму оголошеного предиката бази фактів *dbPredicate*. Застосовувати можна тільки одне з необов'язкових ключових слів. У випадку, коли режим детермінізму предиката явно не заданий – по замовчуванню використовується *nondeterm*, так як у Пролог широко застосовується механізм пошуку з поверненням.

Значення ключових слів:

- *global* - визначає глобальну базу фактів. (Рекомендується замість глобальних фактів застосовувати глобальні предикати, що працюють з локальними фактами).
- *nocopy* – слід використовувати такий підхід слід обережно. Як правило, коли предикат бази фактів викликаний для зв'язування змінної зі строковим або складеним об'єктом, викликані дані копіюються з набору (heap) в глобальний стек Visual Prolog (GStack). *Nocopy* оголошує, що дані не будуть скопійовані, а змінні будуть посилатися безпосередньо на дані факту, що зберігаються у наборі. Це може значно збільшити ефективність, але, якщо копія не була зроблена, після видалення факту, змінна буде вказувати на деяку вже невірну інформацію.
- *nondeterm* – визначає, що база фактів може містити будь-яке число фактів для предиката бази фактів *dbPredicate* – це є режим за замовчуванням.
- *determ* – визначає, що база фактів може містити не більше одного факту для предиката бази фактів *dbPredicate*.
- *single* – визначає, що база фактів завжди містить один і тільки один факт для предиката бази фактів *dbPredicate*.

Зазвичай за своєю природою предикати бази фактів недетерміновані. Якщо жодне з слів *determ* або *single* не використано при оголошенні фактів, компілятор застосовує режим *nondeterm*. Оскільки факти в програму можуть бути додані в будь-який момент виконання, компілятор повинен враховувати, що під час пошуку з поверненням можливо знаходження альтернативних рішень.

Слід з особливою обережністю використовувати детерміновані факти. Оголошення факту детермінованим дозволяє компілятору генерувати більш ефективний код, і при виклику таких предикатів не будете з'являтися попередження про можливий недетермінований виклик. Рекомендовано для таких об'єктів як прапорець, лічильник та подібних. При видаленні факту, який оголошений *determ*, виклик недетермінованих предикатів *retract/1* і *retract/2* буде детермінованим.

Тому, наприклад, якщо відомо, що в будь-який момент часу база фактів містить не більше одного факту *counter*, можна замість

```
facts
counter(integer CounterValue)

predicates
determ retract_d(dbasedom)

clauses
retract_d(X): - retract(X),
!.          % детермінований предикат

goal
retract_d(counter(CurrentCount)), % Пролог не встановить точку відкату
Count= CurrentCount + 1,
asserta(counter(Count)).
```

написати

```
facts
determ counter(integer CounterValue)

goal
retract(counter(CurrentCount)), % Пролог не встановить точку відкату
Count= CurrentCount + 1,
assert(counter(Count)).
```

У випадку застосування ключового слова *single* однократні факти повинні бути відомими, коли програма визиває ціль, тобто вони повинні бути ініційовані у розділах *clauses* у вихідному коді програми, наприклад:

```
facts — properties
single numberWindows_s(integer)

clauses
numberWindows_s(0).
```

Однократні факти не можуть бути видаленими, у випадку такої спроби компілятор генерує помилку. Виклик однократного факту завжди успішний, якщо він викликаний з вільними аргументами, наприклад *numberWindows_s(Num)*, завжди завершується успішно, якщо *Num* є вільною (незв'язаною) змінною.

Використання ключового слова *single* перед декларацією факту дозволяє компілятору виконати оптимізований код для доступу до однократного факту і його модифікації.

Ініціалізація однократних фактів для деяких доменів (для яких не вказано значень за замовчуванням) нетривіальна. Може виявитися корисною наступна інформація:

- бінарні (binary) домени даних можуть бути ініціалізовані шляхом привласнення їм конкретного бінарного значення.

```
global domains font = binary

facts - properties
single my_font(font)

clauses
my_font($[00])
```

- важливим особливим випадком є ініціалізація однократних фактів, що містять стандартний домен *ref*. Домен *ref* є доменом для посилальних чисел у зовнішніх базах даних Visual Prolog, але він також використовується в багатьох зумовлених доменах, оголошених у пакетах, що дозволяє експортувати Visual Prolog. Наприклад, основний домен VPI *window* оголошений так:

```
domains
```

```
window = ref
```

Для ініціалізації значень домену *ref* ви можна використовувати беззнакові числа з попереднім символом тильда

```
facts
single mywin(WINDOW)

clauses
mywin(~0).
```

11.1.2. Оновлення ВБФ

Visual Prolog інтерпретує факти з внутрішньої бази фактів як звичайні предикати, що зберігаються у таблиці, яку можна легко змінити. Тоді як звичайні предикати для досягнення максимальної швидкості компілюються у двійковий код.

Пролог-програму можна розглядати як опис деякої множини відносин – реляційну базу даних. Опис відносин присутній або в явному вигляді (факти), або в неявному вигляді (правила).

Вбудовані предикати дають можливість корегувати таку базу даних (БД) в процесі виконання програми у такий спосіб:

- додаванням нових фактів до програми (в процесі обчислень);
- видаленням з програми вже існуючих фактів.

Перераховані нижче предикати виконують операції над БД, а саме:

- *assert(d)* – завжди успішний і додає факт *d* до бази даних;
- *retract(d)* – видаляє факт, що зіставляється з *d*;
- *retractall(d)* – видаляє всі факти, зіставні з *d*;
- *asserta(d)* – забезпечує запис в базу даних нового факту перед наявними фактами для заданих відносин;
- *assertz(d)* – забезпечує запис в базу даних нового факту після всіх наявних фактів для заданих відносин. Оголошення динамічної бази даних, в яку факти можуть додаватися під час виконання програми або вибиратися з файлу за допомогою предиката *consult*, здійснюється за допомогою ключового слова *database*.

Предикати внутрішньої бази фактів діють аналогічно іншим предикатам тільки з тією різницею, що оголошення таких предикатів розміщені у розділі *facts*.

Предикати у розділі *facts* завжди недетерміновані, тому що факти можуть бути додані у будь-який момент виконання Пролог-програми та існує можливість пошуку альтернативних рішень за допомогою механізму пошуку з поверненням.

11.1.3.Додавання фактів у ВБФ

Предикати, що забезпечують додання нових фактів, можуть використовувати наступні формати:

assert(i)	*assert(i,i)*
asserta(i)	*asserta(i,i)*
assertz (i)	*assertz (i,i)*,

де предикат *assert* працює аналогічно *assertz*.

Так як імена предикатів баз фактів у межах програми або модуля не повторюються, тому точно визначено де саме вставляти факт.

Ціль *assert(людина("Ірина","Жовті Води",18))* забезпечує вставку факту після будь-якого факту предикату *людина*, що зберігається в пам’яті; *asserta(людина("Ірина","Жовті Води",18))* – перед усіма фактами, що зберігаються за предикатом *людина*; *assertz(людина("Ірина","Жовті Води",18))* – після всіх аналогічних фактів предикату *людина*.

Ціль *assert(форма("Ірина","бюджет"),бд_форма))* забезпечує вставку факту про *Ірину* за предикатом *форма*, що зберігається в пам’яті, після будь-якого факту у базі фактів *бд_форма*; *asserta(форма("Ігор","контракт"),бд_форма))* – перед усіма фактами у базі фактів *бд_форма*; *assertz(форма("Ігор","контракт"),бд_форма))* – після всіх аналогічних фактів у базі фактів *бд_форма*.

Приклад використання предиката *assert(i)* (аналогічно *asserta(i) та assertz (i)*):

```
domains
імя, адреса, с_індекс, назва_міста = string
```

```
вік = integer

facts - general
людина (імя, вік, адреса, індекс)
місто(с_індекс, назва_міста)

goal
assert(людина("Ірина", 18, "", 7777)),
assert(людина ("Ігор", 21, "", 5884)),
assert(місто("7777", "Жовті Води")),
assert(місто("5884", "Дніпродзержинськ")).
```

Приклад використання предиката *assert(i,i)*:

```
domains
імя, адреса,с_ індекс, назва_міста = string
вік = integer

facts - с_персон    /* ім'я = с_персон */
людина (імя, вік, адреса, с_індекс)

facts - с_міст      /* ім'я = с_міст */
місто(с_індекс, назва_міста)

goal
assert(людина ("Іван", 20, "", 2605), с_персон),
assert(людина ("Ігор", 19, "", 3031), с_персон),
assert(місто(2605, " Дніпродзержинськ "), с_міст),
assert(місто (3031, " Жовті Води "), с_міст).
```

11.1.4. Видалення фактів з ВБФ

Недетермінований предикат *retract(i)* забезпечує видалення першого факту бази фактів відповідного аргументу, повертаючи у процесі пошуку з поверненням альтернативні рішення і видаляючи всі факти, що ідентифікуються, доки їх взагалі не залишиться – закінчується неуспішно. Роботу *retract(i)* демонструє наступний приклад:

```
domains
list = integer*

facts - dba1
fact1(integer,string,list)

facts - dba2
fact2(integer,string)

clauses
fact1(1,"fact1",[1,2,3]).
fact1(2,"fact2",[1,3]).
fact1(3,"fact2",[3,2,1]).
```

```
fact2(1,"один").
fact2(1,"один ще раз").
fact2(2,"два").
%-------- запити до ВБФ --------

goal
fact1(X,Y,Z).              % запит 1
/*X=1, Y=fact1, Z=[1,2,3]
X=2, Y=fact2, Z=[1,3]
X=3, Y=fact2, Z=[3,2,1]
3 Solutions*/

goal
retract(fact1(X,Y,[_,2|Z])).      % запит 2
/*X=1, Y=fact1, Z=[3]
X=3, Y=fact2, Z=[1]
2 Solutions*/

goal
(fact1(X,Y,Z)).            % запит 3
/*X=1, Y=fact1, Z=[3]
/*X=2, Y=fact2, Z=[1,3]
1 Solution*/

goal
fact1(X,Y,Z).        % запит 4
/*No Solution*/

goal
retract(fact2(1,X)). % запит 5
/*X= один
X= один ще раз
2 Solutions*/
```

Для попередньої бази фактів можна поставити цілі із застосуванням предиката *retract*, що має два аргументи:

```
goal
retract(X,dba1).
/*X=fact1(1,"fact1",[1,2,3])
X=fact1(2,"fact2",[1,3])
X=fact1(3,"fact2",[3,2,1])
3 Solutions*/

goal
retract(fact2(1,X),dba2).
/*X=один
X=один ще раз
2 Solutions*/
```

Ще один приклад застосування предикату у форматі *retract(<факт>,[ім'я ВБФ])*:

```
facts
людина(string,string)
facts-бд_подобається
подобається(string,string)
не_подобається(string,string)

clauses
людина("Ігор Рось","Олександрія").
людина("Іван Дрига","Жовті Води").
людина("Марк Глєб","Дніпродзержинськ").

подобається("Ігор Рось","книги").
подобається("Іван Дрига","велоспорт").
подобається("Марк Глєб","програмування").
подобається("Ігор Рось","біг").
подобається("Іван Дрига","плавання").

не_подобається("Ігор Рось","плавання").
не_подобається("Марк Глєб","плавання").
не_подобається("Марк Глєб","гребля").
```

Ціль *retract(людина("Ігор Рось",_))* видаляє зі стандартної бази даних *facts* перший факт «*людина*», що стосується «*Ігор Рось*».

Ціль *retract(подобається(_,"плавання"))* видаляє з ВБФ *бд_подобається* перший факт, що відповідає *подобається(X, "плавання")*. Імена предикатів баз даних не повторюються: предикат *людина* збережений у стандартній БД, а *подобається* – тільки у БД *бд_подобається*.

Ціль *retract(подобається(_,"книги"))* демонструє використання необов'язкового другого аргументу для контролю типів – мета вважається досягнутою, якщо з БД *бд_подобається* видаляється перший факт, що відповідає *подобається(_,"книги")*.

Ціль *retract(студент("Марк Глєб",_),бд_подобається)* виводить повідомлення про помилку типу.

За допомогою *retract* можна видалити всі факти з бази даних. Якщо викликати з вільною змінною у якості першого аргументу, то для уточнення бази даних, з якої потрібно видалити факти, підлягатиме аналізу другий аргумент, наприклад:

```
goal
retract(X,dba1),
write(X),
```

```
fail.
```

Для одночасного видалення усіх фактів з бази даних використовується предикат *retractall*. Цей предикат успішний, виконується тільки один раз та не повертає вихідних значень. У середині предиката не можуть використовуватися вільні (непов'язані) змінні, такі змінні замінюють символом підкреслення.

```
domains
list = integer*

facts - dba1
fact1(integer,string,list)

facts - dba2
fact2(integer,string)

clauses
fact1(1,"fact1",[1,2,3]).
fact1(2,"fact2",[1,3]).
fact1(3,"fact2",[3,2,1]).
fact2(1,"один").
fact2(1,"один ще раз").
fact2(2,"два").
```

Запит до ВБФ *fact1*

```
goal
fact1(X,Y,Z).
/*X=1, Y=fact1, Z=[1,2,3]
X=2, Y=fact2, Z=[1,3]
X=3, Y=fact2, Z=[3,2,1]
3 Solutions*/
```

Видалити всі записи ВБД *fact1*

```
goal
retractall(fact1(_,_,_)).      % завжди успішний
%Yes
```

Після повторного запиту про факти ВБФ *fact1* рішення відсутні (факти видалені).

```
goal
fact1(X,Y,Z).
%No Solution
```

Складена ціль видаляє всі факти з ВБФ *fact2*, що співпадають із запитом:

```
goal
retractall(fact2 (1,_)).
%Yes
```

Предикат у форматі retractall(<факт>,ім'яВБФ) видаляє факти з ВБФ dba1:

```
goal
retractall(_,dba1)
Yes
```

Перевірка наявності фактів у *dba1* не має рішення.

```
goal
fact1(X,Y,Z)
No Solution
```

Факти, визначені під час компіляції в розділі *clauses*, також можуть бути видалені, вони нічим не відрізняються від фактів, доданих під час виконання.

Слід остерігатися випадкового написання коду, який двічі стверджує один і той же факт. Внутрішні бази фактів не передбачають ніякої унікальності. Один і той же факт може з'являтися у внутрішній базі даних фактів багато разів. Однак, версію *assert* з перевіркою на унікальність написати дуже просто:

```
facts - люди
персона (string,string)

predicates
uassert(люди)

clauses
uassert(персона(Name,Address)):-
персона(Імя,Адреса),
!,
  %;   або
assert(персона (Імя,Адреса)).
```

11.1.5. Читання нових фактів під час виконання програми

Предикат *consult* призначений для читання нових фактів під час виконання програми. *Consult* зчитує з файлу *fileName* факти, описані в розділі *facts*, і вставляє їх у програму в кінець відповідної бази фактів. Предикат *consult* визивається з одним або двома аргументами:

```
consult(fileName)   % (i) з одним аргументом
consult(fileName, databaseName)  % (i, i) з двома аргументами
```

Однак на відміну від предиката assertz, якщо викликати *consult* тільки з одним аргументом (без імені бази фактів), то будуть зчитані лише факти, які були описані в розділі без імені (за замовчуванням *dbasedom*).

```
domains
ilist=integer*

facts - general
```

```
p1(integer,char,real,string,symbol,ilist)
p2(integer)

goal
consult("File1.dba").
/* -------- вміст " File1.dba" може бути -------
p1(1,'a',44.44,"PC","Prolog",[1,2,3,4])
p1(2,'b',-4.444E-98,"---","++++",[ ])
p2(88)
p2(99)
……..*/
```

Якщо викликати *consult* з двома аргументами (ім'я файлу і ім'я бази фактів), то будуть перевірені тільки факти з вказаної бази фактів.

```
domains
ilist=integer*

facts - facts1
p1(integer,char,real,string,symbol,ilist)

facts - facts2
p2(integer)

goal
consult("facts1.dba",facts1).
consult("facts2.dba",facts2).
/* -------- Вмісm facts1.dba --------
p1(1,'a',44.44,"Visual","Prolog",[1,2,3,4])
p1(2,'b',-4.444E-98,"---","++++",[])
……*/
/* -------- Вмісm facts2.dba --------
p2(88)
p2(99)
……*/
```

Якщо файл містить що-небудь ще, крім фактів зазначеної бази, то предикат *consult*, коли він дійде до цього рядка, поверне помилку. Слід пам'ятати, що предикат *consult* зчитує по одному факту. Якщо файл містить десять фактів, а в сьомому факті є яка-небудь синтаксична помилка, *consult* занесе шість перших фактів в базу даних фактів, після чого виведе повідомлення про помилку.

Предикат *consult* може зчитувати файли тільки у тому форматі, який створює предикат *save*. Файли не повинні містити:

- символів верхнього регістру, за винятком тих, що містяться всередині рядків у подвійних лапках;

- пропусків, за винятком тих, що містяться всередині рядків у подвійних лапках;
- коментарів;
- порожніх рядків;
- ідентифікаторів (*symbol*) без подвійних лапок.

При створенні або зміні файлу з фактами в редакторі потрібно бути уважними.

11.1.6. Збереження бази фактів під час роботи програми

Предикат *save* призначений для збереження фактів з вказаної бази фактів (розділу *facts*) у файлі. Формат запису *save* наступний:

```
save(fileName) % (i) з одним аргументом
save(fileName,databaseName) % (i,i) з двома аргументами
```

Під час виклику предиката *save* з одним аргументом (без уточнення імені бази фактів), у файлі *fileName* будуть збережені факти з бази фактів *dbasedom*, яка використовується по замовчуванню. Наприклад, збереження файлу з ім'ям «*con*»:

```
facts - general
fact1(integer,string,list)
fact2(integer,string)

clauses
fact1(1,"fact1",[1,2,3]).
fact1(2,"fact2",[1,3]).
fact1(3,"fact2",[3,2,1]).

fact2(1,"один").
fact2(1,"один ще раз").
fact2(2,"два").
%-------- запити --------

goal
save("con").

/*fact1(1,"fact1",[1,2,3])
fact1(2,"fact2",[1,3])
fact1(3,"fact2",[3,2,1])
fact2(1,"один")
fact2(1,"один ще раз")
fact2(2,"два")
Yes*/
```

Виклик предиката *save* з двома аргументами (ім'я файлу та ім'я бази фактів), у вказаному файлі будуть збережені факти з розділу *facts* бази фактів з іменем *databaseName*. Наприклад:

```
domains
list = integer*

facts - dba1
fact1(integer,string,list)
facts - dba2
fact2(integer,string)

clauses
fact1(1,"fact1",[1,2,3]).
fact1(2,"fact2",[1,3]).
fact1(3,"fact2",[3,2,1]).

fact2(1,"один").
fact2(1,"один + один").
fact2(2,"два").
/*для переконання слід перевірити ціль
Goal
save("con",dba1).
Збережені дані будуть відображатися у діалоговому вікні */
%---------- запити ----------
goal
save("con",dba1).

/*fact1(1,"fact1",[1,2,3])
fact1(2,"fact2",[1,3])
fact1(3,"fact2",[3,2,1])
Yes*/

goal
save("dba1 dba",dba1).
%Yes

goal
save("dba2 dba", dba2).
%Yes
```

11.2. Зовнішні бази даних Visual Prolog

Мова Пролог оптимально пристосована для роботи з реляційними базами даних.

Постійно зростаючий обсяг інформації, яку необхідно обробляти сучасним комп'ютерам пред'являє більш широкі вимоги до сучасних баз даних. Бази даних зараз використовуються не тільки як звичайні сховища інформації, але і

як сховища знань. Тому з'являється нова вимога до баз даних, яка прийшла від баз знань, – це можливість логічного висновку нових знань із вже відомих, а також робота в режимі експертної системи, що широко застосовують систему внутрішніх баз фактів.

При використанні ВБФ слід зазначити важливий недолік: перший – база фактів зберігається в текстовому файлі у форматі мови Пролог, який завантажується в пам'ять, і компілюється під час виконання програми; другий – Прологу доводиться завантажувати в пам'ять всю базу фактів, тобто, якщо оперативної пам'яті комп'ютера не вистачає, то програма не може працювати.

Підключення до Прологу універсальних баз даних дозволяє зняти ці два обмеження. Використання баз даних дозволяє зняти обмеження на об'єм оперативної пам'яті комп'ютера, так як менеджер баз даних завантажує в пам'ять тільки ті дані, які потрібні в даний момент, а не все відразу. Також не потрібна компіляція фактів під час виконання програми, так як вони вже знаходяться в потрібному форматі. Таким чином, збільшується швидкість і якість введення знань. Більш того, знання можуть додаватися прямо по ходу виконання Пролог-програми.

Стає можливим надходження даних відразу з різних комп'ютерів. Використання баз даних дозволяє працювати з однією базою знань декількома програмами, а також надається можливість зручного редагування бази з допомогою інших програм. Завантаження та пошук записів в БД покладено на операційну систему, яка централізовано і ефективно розподіляє доступ до баз даних для кількох програм, а також за рахунок вбудованого кешування дозволяє знизити залежність швидкості виконання програми від швидкості роботи диска.

Звернення до бази даних відбувається так само, як до звичайних предикатів, що складається тільки з фраз (фактів і правил).

При запуску Пролог-програми інтерпретатор визначає, чи існує файл бази даних на диску. Якщо файл бази даних існує, то він відкривається та перевіряється відповідність структури файлу бази даних і опису цієї бази даних

в програмі. Якщо файл бази даних відсутній на диску, то він створюється за описом структури. При завершенні виконання Пролог-програми всі бази даних автоматично закриваються.

Предикати системи зовнішніх баз даних Visual Prolog забезпечують:

- ефективну обробку дуже великих об'ємів даних, що зберігаються на диску;
- розміщення баз даних у файлах, пом'яті і розширеній пам'яті типу EMS;
- більш гнучку обробку даних, порівняно з тією, що забезпечує механізм автоматичного пошуку з поверненням у Пролог;
- збереження у файлах і завантаження зовнішніх баз даних у двійковому коді.

11.2.1. Структура зовнішніх БД

Системи управління базою даних (СУБД) орієнтовані на діалог з користувачем. Будь-яка система такого роду повинна містити, як мінімум, такі можливості:

- занесення в базу нових даних;
- видалення даних з бази;
- вибірка і висновок містяться в базі даних.

Ці вимоги передбачають наявність у системі меню, яке представляє користувачеві можливість легко орієнтуватися при зверненні до стандартних функцій СУБД, а також віконної системи, дає чітке уявлення про доступні користувачеві засоби.

Як приклад структурної схеми БД може бути наступна (рис. 11.1).

Схема показує, що модуль Меню дозволяє користувачеві вибрати між чотирма модулями: *process*(1) для запису даних у базу, *process*(2) для видалення даних, *process*(3) для виведення даних на екран, і *process*(4) для виходу з системи.

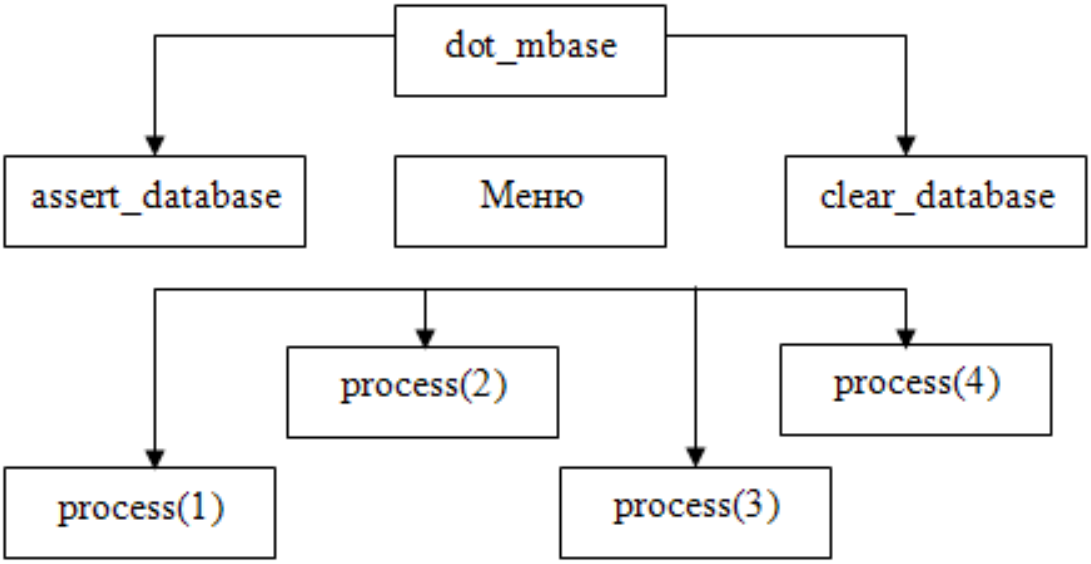


Рисунок 11.1. Структурна схема БД

Зовнішня БД складається з елементів даних, що зберігаються у ланцюжках та відповідних бінарних деревах B+, що використовують для швидкого доступу до даних.

Для простих операцій додавання нових елементів до бази даних, заміна або видалення існуючих елементів бінарні дерева не використовуються. Вони використовуються при реалізації задач упорядкування або пошуку елементів у базі даних.

Особливості запису імен стандартних предикатів управління базами даних наступні: перша частина імені (*db_*, *chain_*, *term_* та ін.) ідентифікує вхідні дані, наступна (*flush, btrees, delete* та ін.) – указує на вихідні. Наприклад *term_ delete* видаляє терм.

Пролог підтримує роботу з декількома одночасно відкритими зовнішніми БД. Для їх розпізнавання у кожному виклику стандартного предиката зовнішніх баз даних використовуються селектори, які повинні бути оголошені в домені *db_selector* (аналогічний домену *file* файлової системи), наприклад *db_selector=склади; матеріали*.

Зовнішня база даних Пролог складається з сукупності термів, що зберігаються у ланцюжках. База даних може налічувати різну кількість ланцюжків, які в свою чергу можуть містити різну кількість термів.

Відносини в БД і таблиці предикатів моделюються ланцюжками термів. Як приклад, можна розглянути дані обліку товарів, що містять інформацію про постачальників, споживачів та самі товари, відносини про які треба помістити в одну базу даних – три ланцюжки, що іменуються «постачальники», «споживачі» та «товари» відповідно. Щоб вставити терм до зовнішньої БД треба вставити його в іменований ланцюжок. Для пошуку терма без іменованого ланцюжка, потрібно вказувати домен, якому належить терм. Рекомендовано, щоб всі терми у ланцюжку відповідали одному домену. На розміщення термів в базі даних обмеження не накладаються.

Наступний приклад демонструє формування двох ланцюжкових баз даних: *dba1* та *dba2:*

```
domains
db_selector = dba1; dba2
споживачі = споживач(імя_спож, адреса)
вироби = виріб(назв_вир, к, імя_спож)
імя_спож, назв_вир = symbol
     % шифр
к = integer
адреса = string

predicates
доступ - procedure ()

clauses
доступ:-
   chain_terms(dba1,"ланцюжок1", споживачі, споживач (Імя,Адреса),_),
   chain_terms(dba2," ланцюжок2", вироби, виріб (Виріб,К,Імя),_),
   write("відправити ",Виріб," шифр виробу ", К, Адреса),
   nl,
   fail.

доступ.

goal
% створити БД: dba1 та dba2
db_create(dba1,"dd1",у_памяті),
db_create(dba2,"dd1.bin",у_файлі),

% вставити факти по споживачах ланцюжок1 в dba1
chain_insertz(dba1,"ланцюжок1",споживачі,
споживач("Іван Мазур","Кропоткіна 123"),_),
chain_insertz(dba1,"ланцюжок1",споживачі,
споживач("Ігор Крутий","Франка 31"),_),
chain_insertz(dba1," ланцюжок1", споживачі,
```

```
споживач ("Сергій Бор","Б.Хмельницького 23"),_),
chain_insertz(dba1," ланцюжок1", споживачі,
споживач ("Дмитро Білий","Франка 123"),_),

% вставити факти за вибором ланцюжок2 в dba2
chain_insertz(dba2," ланцюжок2",вироби,
виріб ("стіл 4452",231," Ігор Крутий "),_),
chain_insertz(dba2," ланцюжок2",
вироби, виріб ("стіл 3347",331," Сергій Бор "),_),
доступ,
db_close(dba1),
db_close(dba2),
db_delete("dd1",у_памяті),
db_delete("dd1.bin",у_файлі).
```

У *dba1* наведеного прикладу зберігаються всі дані про споживача, у *dba2* – про вироби. На початку Пролог-програма створює бази даних *dba1* у пам'яті комп'ютера та *dba2* у файлі. Далі факти вставляються у два ланцюжки, відповідно chain1 та chain2. Після ланцюжки переглядаються на предмет виявлення споживача і замовленого ним виробу. Якщо така пара знаходиться, то повертається адреса, за якою постачальник повинен доставити виріб. На завершу вальному етапі програма виконує закриття та видалення обох баз даних.

11.2.2. Типи зовнішніх БД

Бази даних використовують шість стандартних доменів, це:

- *db_selector* – оголошує селектори баз даних;
- *bt_selector* – оголошує селектори бінарних дерев;
- *place* – оголошує місцезнаходження БД у пам'яті чи на диску;
- *accessmode* – вирішує як використовувати файл;
- *denymode* – визначає як користувачі будуть відкривати файл;
- *ref* – посилається на місцезнаходження терма у ланцюжку.

При введені нового терма у БД Visual Prolog йому привласнює *число-покажчик*, який можна використовувати для вибірки, переміщення або заміни терма. Якщо це B+ дерево, то число-покажчик використовується для сортування чи пошуку терма.

Покажчик *ref*, що вказує посилання на місцезнаходження терма, не можна ввести з клавіатури. Аргументи предикатів, призначених для роботи з

покажчиком, відносяться до домену *ref*. При видаленні предикатом *term_delete*, система знову зможе автоматично використати покажчик віддаленого терму при внесені нового терма у БД. Стандартний предикат *db_reuserefs* перевіряє помилки у випадку, коли покажчик був збережений у внутрішній базі фактів або в B+ дереві. *db_reuserefs* має формат:

db_reuserefs(Dbase,ReUse) %(i,i) два аргумента

Dbase – це *db_selector, ReUse* – беззнакове ціле число ( ) – для перевірки та 1 – для заборони перевірки.

11.2.3. Обробка БД

При створенні або відкритті БД її можна помістити у файл або в оперативну пам'ять у залежності від значення аргументу *place* при зверненні до *db_create* або *db_open*. Закриття БД відбувається при визові *db_close (*табл. 11.1).

Таблиця 11.1. Предикати обробки БД

Предикат	Призначення
1	2
db_create (i,i,i)	Предикат призначений для створення нової БД. Формат: *db_create(Dbase, Name, Place)* *Name* – вказує ім'я файлу (при створенні БД на диску); використовується у якості параметра *db_open* та *db_close (*при створенні в оперативній пам'яті). *Place*– вказує місце розміщення БД, вказує на розміщення: *in_file*(на диску), *in_memory*(в оперативній пам'яті), *in_ ems*(в EMS розширеної пам'яті). Оголошується у розділі *domains* у форматі *Place= in_file; in_memory; in_ ems*
db_open (i,i,i)	Предикат призначений для відкриття збереженої БД. Формат: *db_open (Dbase, Name, Place)* *Name* – вказує ім'я файлу(коли *Place* приймає значення *in_memory);* відповідає значенню файла, прийнятому у DOS*(*коли *Place* приймає значення *in_file).*

Продовження таблиці 11.1

db_copy (i,i,i)	Предикат призначений для копіювання збереженої БД. Формат: *db_copy (Dbase, Name, Place)* Створити копію БД можна незалежно від того, де розміщена БД. Предикат *db_copy* використовується у випадках: для збереження БД в оперативну пам'ять та для збереження її у файлі після обробки(замість *save* та *consult*); для копіювання з диску в пам'ять; для стиснення БД.
db_openinvalid (i,i,i)	Предикат призначений для відкриття пошкодженої БД. Формат: *db_openinvalid (Dbase, Name, Place)* Можливе поновлення частини пошкоджених даних.
db_flush (i)	Предикат призначений для звільнення буфера та запису їх вмісту у місця призначення у БД. Формат: *db_ flush (Dbase)* Після поновлення БД вона залишається несправною до запису за допомогою *db_ flush* або до закриття.
db_close (i)	Предикат призначений для закриття БД. Формат: *db_ close(Dbase)*, де *Dbase* – ім'я БД.
db_delete (i,i)	Предикат призначений для видалення БД, якщо вона розміщена у файлі та визволяє пам'ять, якщо БД розміщена в оперативній пам'яті. Формат: *db_ close(Dbase, Place)*, де *Dbase* – ім'я БД.
db_btrees (i,o)	Предикат призначений для послідовного зв'язування *BtreeName* з іменем кожного B+ дерева у БД. *db_btrees(Dbase, BtreeName)*
db_chains (i,o)	Предикат призначений для почергового зв'язування змінної *ChainName* з іменем кожного ланцюжка БД. *db_chains(Dbase, ChainName)*

Продовження таблиці 11.1

1	2
db_statistics (i,o,o,o,o)	Виведення статистичних відомостей про БД. Формат: *db_statistics(Dbase, NoOfTerms, MemSize, DbaSize, FreeSize)*,

	де *Dbase* – ім'я БД; *NoOfTerms* – загальне число термів у БД; *MemSize* – розмір в байтах внутрішньої таблиці для БД, що зберігається у пам'яті; *DbaSize* – загальна кількість байт, зайнятих термами та визначеннями у БД; *FreeSize* – величина вільної пам'яті, що залежить від розміщення БД в поточний час.

Приклад використання предикату *db_selector*

```
domains
db_selector = mydba

goal
db_create(mydba,"dd.bin",y_файлі),
%предикати оновлення та доступу до БД
db_close(mydba),
db_delete("dd.bin",y_файлі).
```

Приклад використання предикату *db_open*

```
domains
db_selector = mydba

goal
db_open(mydba,"DD.BIN",y_файлі),
   %доступ до БД
db_close(mydba).
```

або

```
domains
db_selector = mydba

goal
db_open(mydba,"share.dba",читання,заборона_запису),
/* оновлення і доступ до бази даних, всередині операцій*/
db_close(mydba).
```

Приклад використання предикату *db_copy*

```
domains
db_selector = mydba

goal
db_open(mydba,"\\prolog\\programs\\register.bin",in_file),
db_copy(mydba,"реєстр",y_памяті),
db_close(mydba),
db_open(mydba," реєстр",y_памяті),
   % Продовжити роботу з БД у пам'яті
db_copy(mydba,"dd.bin",in_file),
```

```
db_close(mydba),
db_delete("реєстр",y_памяті).
```

Приклад використання предикату *db_openinvalid*

```
domains
db_selector = mydba
dbdom = місто(місто_в,назва);
особа(імя,прізвище,вулиця, місто_в,код)
місто_в,назва, імя,прізвище,вулиця, місто_в,код = string

goal
db_openinvalid(mydba,"\\prolog\\programs\\register.bin", y_файл),
db_chains(mydba,Ланцюжок),% у текстовий файл
chain_terms(mydba, Ланцюжок,dbdom,Term,Ref),
write("\n Терм=",Term,", Посилання=",Ref),fail.
```

Приклад використання предикату *db_flush*

```
domains
db_selector = mydba

goal
db_open(mydba,"\\prolog\\programs\\register.bin",y_файл),
% виконати деякі оновлення
db_flush(mydba).
```

Приклад використання предикату *db_close*

```
domains
db_selector = mydba

goal
db_open(mёydba,"dd.bin",y_файл),
%Оновлення та доступ до БД
db_close(mydba).
```

Приклад використання предикату *db_delete*

```
domains
db_selector = mydba

goal
db_open(mydba,"dd.bin",y_память),
% Оновлення та доступ до БД
db_close(mydba),    % БД існує
db_delete("dd.bin",y_память).
```

Приклад використання предикату *db_btrees*

```
domains
db_selector = mydba

goal
db_open(mydba,"\\prolog\\programs\\register.bin",y_файл),
db_btrees(mydba,BtreeName), % ім'я B-дерева
write(BtreeName),nl,fail.
```

Приклад використання предикату *db_chains*

```
domains
db_selector = mydba

goal
db_open(mydba,"\\prolog\\programs\\register.bin",y_файл),
db_chains(mydba,ChainName),
write(ChainName), % ім'я ланцюжка
nl,fail.
```

Приклад використання предикату *db_statistics*

```
domains
db_selector = mydba

goal
db_open(mydba,"\\prolog\\programs\\register.bin",y_файл),
db_statistics(mydba,ВідсТерми,РозмПамяті,РозмБД,ВільнРозм),
write("\n\nЗагальне число записів у БД:", ВідсТерми),
write("\nКількість байт, що використовується в основній пам'яті:",РозмПамяті),
write("\nКількість байт у БД, що використовується:", РозмБД),
write("\nКількість вільних байтів на диску:",ВільнРозм).
```

Для вставки термів у ланцюжок зовнішньої БД використовуються предикати *chain_inserta*, *chain_insertz*, *chain_insertafter*.

Можна послідовно зв'язувати терми у ланцюжку і їх покажчики з аргументами предиката *chain_terms*, а предикат *chain_delete* дає змогу видаляти із зовнішньої БД цілий ланцюжок термів (табл. 11.2).

Таблиця 11.2. Предикати обробки ланцюжків

Предикат	Призначення
1	2
chain_inserta (i,i,i,i,o)	Предикат відповідний asserta, включає *Term* на початок ланцюжка *Chain*. Формат: *chain_inserta (Dbase, Chain, Domain, Term, Ref)* *Dbase* – це db_selector БД. *Domain* – тип змінної *Term*. *Ref* – це покажчик на відповідний *Term*.

Продовження таблиці 11.2.

1	2
chain_insertz (i,i,i,i,o)	Предикат відповідний assertz, включає *Term* в кінець ланцюжка *Chain*. Формат: *chain_inserta (Dbase, Chain, Domain, Term, Ref)*
chain_insertafte	Предикат призначений для розміщення(вставки) терма після

r (i,i,i,i,i,o)	елемента ланцюжка, що визначається покажчиком *Ref*. зв'язується з покажчиком, що відповідає терму *Term* після його вставки. Формат: *chain_insertafter(Dbase, ChainName, Domain, Ref, Term, NewRef)*
chain_terms (i,i,i,o,o)	Предикат призначений для зв'язування з кожним термом *Term* та *Ref*, відповідним йому, покажчиком в ланцюжку *Chain.* Формат: *chain_terms(Dbase, Chain, Domain, Term, Ref)*
chain_delete (i,i)	Предикат призначений для видалення визначеного ланцюжка. Формат: *chain_delete(Dbase, Chain)*
chain_first (i,i,o)	Предикат призначений для повернення першого елемента ланцюжка. Формат: *chain_first(Dbase, Chain, FirstRef)*
chain_last (i,i,o)	Предикат призначений для повернення останнього елемента ланцюжка. Формат: *chain_last(Dbase, Chain, LastRef)*
chain_next (i,i,o)	Предикат призначений для повернення покажчика терма, наступного за поточним у ланцюжку. Формат: *chain_next(Dbase, Ref, NextRef)*
chain_prev (i,i,o)	Предикат призначений для повернення покажчика терма, попереднього поточному терму у ланцюжку. Формат: *chain_ prev(Dbase, Ref, NextRef)*

Приклад використання предикату *chain_inserta*

```
domains
db_selector = mydba
dbdom = н(integer,char,real,string)

goal
db_open(mydba,"dd.bin",y_файл),
chain_inserta (mydba,н_ланцюжок,dbdom, н(1,'x',88.99,"Visual Prolog"),_),
db_close(mydba).
```

Приклад використання предикату chain_first та chain_insertafter

```
domains
db_selector = mydba
```

```
dbdom = н(integer,char,real,string)

goal
db_open(mydba,"dd.bin",у-файл),
chain_first(mydba,н_ланцюжок,ПершеПосилання),
chain_insertafter(mydba, н_ланцюжок,dbdom, ПершеПосилання,
н(1,'x',88.99,"Visual Prolog"),_),
db_close(mydba).
```

Приклад використання предикату chain_terms

```
domains
db_selector = mydba
dbdom = city(cityno,cityname);
person(firstname,lastname,street,cityno,code)
cityno, cityname, firstname, lastname, street, code = string

goal
db_open(mydba,"\\prolog\\programs\\register.bin",in_file),
db_chains(mydba,Chain),
chain_terms(mydba,Chain,dbdom,Term,Ref),
write("\nTerm=",Term,", Ref=",Ref),
fail.
```

Приклад використання предикату chain_delete

```
domains
db_selector = mydba

goal
db_open(mydba,"ddr.bin",у_файл),
chain_delete(mydba,"ім'я ланцюжка").
```

У Visual Prolog існують предикати, за допомогою яких обробляються терми. При звернені до *term_replace*, *term_delete* і *ref_term* необхідно оголосити домен терма у якості аргументу (табл. 11.3).

Рекомендується оголошувати всі терми БД як альтернативи одного домена, у зовнішній БД немає обмежень на змішування доменів.

Таблиця 11.3. Предикати обробки термів

Предикат	Призначення
term_replase (i,i,i,i)	Предикат призначений для заміни терма з покажчиком БД *Ref* на новий з іменем *Term*. Формат: *term_replase (Dbase, Domain, Ref , Term)* *Dbase* – це ім'я БД. *Domain* – тип змінної *Term*. *Ref* – це покажчик на відповідний *Term*.
term_delete (i,i,i)	Предикат призначений для видалення терма з покажчиком *Ref*. Формат: *term_delete (Dbase, Chain, Ref)*

ref_term (i,i,i,o)	Предикат призначений для зв'язування *Term* з термом, розміщеним під покажчиком *Ref*. Формат: *ref_term(Dbase, Domain, Ref, Term)*

Приклад використання предикату *term_replase*

```
domains
db_selector = mydba
dbdom = н(integer,char,real,string)

goal
db_open(mydba,"dd.bin",y_файл),
chain_first(mydba,н_ланцюжок,ПершийПокажчик),
term_replace(mydba,dbdom, ПершийПокажчик,
н(1,'H',60.55,"Visual Prolog")),
db_close(mydba).
```

Приклад використання предикату *term_delete*

```
domains
db_selector = mydba
dbdom = f(integer,char,real,string)

goal
db_open(mydba,"dd.bin",y_файл),
chain_first(mydba, н_ланцюжок, ПершийПокажчик),
term_delete(mydba, н_ланцюжок, ПершийПокажчик),
db_close(mydba).
```

Приклад використання предикату *ref_term*

```
domains
db_selector = mydba
dbdom = місто(місто_н,назва);
особа(імя,прізвище,вулиця,місто_н,індекс)
назва, імя,прізвище,вулиця,місто_н,індекс = string
predicates
rd(Ref)

clauses
rd(Ref) :-ref_term(mydba,dbdom,Ref,Term),
write(Term),nl, fail.
rd(Ref) :-chain_next(mydba,Ref,Next),!,
rd(Next).
rd(_).

goal
db_open(mydba,"\\prolog\\programs\\register.bin",y_файл),
db_chains(mydba,Ланцюжок),
chain_first(mydba, Ланцюжок,Ref),
rd(Ref). % покажчик на відповідний Term
```

11.2.4.Дерева B+

B+ дерева подібні бінарним деревам. Але для перших у кожному вузлі запам'ятовується більше одного ключового рядка. Шляхи пошуку у B+ дереві для кожного ключа у «листях» дерева мають одну й ту ж довжину.

У Visual Prolog дерева зберігаються у зовнішніх БД та забезпечують *індексування* БД. Кожен вхід у B+ дерево відповідає *ключовому рядку* та, зв'язаному з ним, *покажчику* бази даних. Для створення БД спочатку треба створити *запис* та визначити для нього ключ, які далі будуть включені у B+ дерево. B+ дерева не вимагають покажчиків на терми у тій же БД. Пролог дає можливість мати БД, що містить ланцюжки на іншу БД з B+ деревом, яке вказує на терми у цих ланцюжках.

У B+ деревах ключі групуються в *сторінки*. Кожна сторінка має однаковий розмір. Всі сторінки можуть містити однакову кількість ключів, однаковий розмір ключів, що зберігаються в одному дереві B+. Розмір ключів визначається параметром *ДовжКлюча*, який необхідно вказувати, створюючи дерево B+. При спробі вставити в дерево B+ рядки, що перевищують *ДовжКлюча*, Пролог усікатиме їх до встановленої довжини.

При створенні дерева B+ потрібно також задавати параметр *Порядок*, який визначає кількість ключів, що повинні зберігатися у кожному вузлі дерева (від 4 до 8 ключів у кожному вузлі). При формуванні дерева B+ потрібно передбачити появу усіх дублікатів ключів.

У мові Visual Prolog є предикати для роботи з B+ деревами (табл. 11.4).

Таблиця 11.4. Предикати дерев B+

Предикати	Призначення
1	2
bt_create (i,i,o,i,i) та bt_create(i,i,o,i,i,i)	Предикат призначений для створення нового дерева. Формат: *bt_create (Dbase, BtreeName, Btree_Sel, KeyLen, Order)* *BtreeName* –ім'я нового дерева. *bt_create (Dbase, BtreeName, Btree_Sel, KeyLen, Order, Duplicates*
bt_open	Предикат відкриває створене дерево Формат:

(i,i,o)	*bt_open(Dbase, BtreeName, Btree_Sel)*
bt_close (i,i) bt_delete (i,i)	Предикати призначені для закриття/видалення дерева B+ Формат*: bt_close(Dbase, Btree_Sel)* *bt_delete(Dbase, BtreeName)*
bt_copyselector (i,i,o)	Предикат створює новий покажчик на вже відкритий *bt_copyselector(Dbase, OldBtree_sel, NewBtree_Sel)*
bt_statiostics (i,i,o,o,o,o,o,o)	Предикат повертає статистичну інформацію *bt_statiostics*(*Dbase,Btree_Sel,NumKeys,NumPages,Depth,KeyLen, Order,PgSize)*
key_insert (i,i,i,i) key_delete (i,i,i,i)	Предикати призначені для оновлення B+ дерева *key_insert(Dbase, Btree_Sel, Key, Ref)* *key_delete(Dbase, Btree_Sel, Key, Ref)*
key_first (i,i,o) key_last (i,i,o) key_search (i,i,i,o); (i,i,i,i);	Предикати встановлюють покажчик на перший, останній вузол та на даний ключ *key_first(Dbase, Btree_Sel, Ref)* *key_last(Dbase, Btree_Sel, Ref)* *key_ search(Dbase, Btree_Sel, Key, Ref)*
key_next (i,i,o) key_prev (i,i,o)	Предикати використовуються для переміщення покажчика (вперед, назад). Формат: *key_next(Dbase, Btree_Sel, NextRef)* *key_prev(Dbase, Btree_Sel, PrevRef)*

Продовження таблиці 11.4

1	2
key_current (i,i,o,o)	Предикат повертає ключ з покажчиком БД, який відповідає поточному положенню покажчика B+ дерева *key_current(Dbase, Btree_Sel, Key, Ref)*

11.2.5. Методи роботи з системою зовнішніх баз даних Visual Prolog

Використовуються наступні методи роботи з зовнішніми БД: послідовний перегляд ланцюжків або B+ дерев; виведення вмісту; захист інформації; внесення змін; установка покажчиків всередині відкритого B+ дерева; зміна структури.

Наступний приклад демонструє роботу предиката *bt_keys*, який повертає кожен ключ в B+ дереві та зв'язаний з ним покажчик БД. Для виведення поточного стану БД використовується предикат *listdba*, в якості аргументу якого виступає селектор відкритої БД.

```
constants
ifdef os_unix
імяФайлу = "/usr/local/PDCProlog/programs/register/register.bin"
elsedef
ifdef os_os2
імяФайлу = "c:\\vip74\\vpi\\programs\\register\\exe\\register.bin"
elsedef
імяФайлу = "c:\\Program Files\\vip74\\vpi\\programs\\register\\exe\\register.bin"
enddef
enddef

domains
db_selector = мояБД
проживання = місто(індекс, назва);
особа(імя, прізвище, вулиця, індекс, код)
індекс, назва, імя, прізвище, вулиця, код = string

predicates
списокБД(db_selector) - procedure (i)
bt_keys(db_selector,bt_selector,string,ref) - nondeterm (i,i,o,o)
bt_keysloop(db_selector,bt_selector,string,ref) - nondeterm (i,i,o,o)

clauses
списокБД(Db_selector):-nl,
   write("*********************************************"),nl,
   write("         лістинг БД              "),nl,
   write("*********************************************"),
db_statistics(Db_selector,NoOfTerms,MemSize,DbaSize,FreeSize),nl,nl,
write("Загальне число записів в базі даних:",NoOfTerms),nl,
write("Кількість байт, використовуваних в основній пам'яті:", MemSize),nl,
write("Кількість байт, використовуваних в базі даних:",DbaSize),nl,
write("вільний об'єм:",FreeSize),nl,
   fail.

списокБД(Db_selector):-
   db_chains(Db_selector,Chain),nl,nl,nl,nl,
   write("******* Корегувати лістінг **********"),nl,nl,
   write("Ім'я=",Ланцюг),nl,nl,
   write("зміст:",Ланцюг),nl,
   write("----------------------------\n"),
   chain_terms(Db_selector, Chain, mydom,Term, Ref),
   write("\n", Ref, ":",Term), fail.

списокБД(Db_selector):-
   db_btrees(Db_selector,Btree),
```

```
    bt_open(Db_selector,Btree,Bt_selector),
    bt_statistics(Db_selector,Bt_selector,NoOfKeys,
          NoOfPages,Dept,KeyLen,Order,PageSize),nl,nl,nl,
    write("******** список індексів **************"),nl,nl,
    write("Ім'я=", Btree),nl,
    write("Кількість клавіш=", NoOfKeys),nl,
    write("Сторінок=", NoOfPages),nl,
    write("Відділ=", Dept),nl,
    write("Замовлення=", Order),nl,
    write("Ключ=", KeyLen),nl,
    write("Розмір сторінки=", PageSize), nl,
    write("зміст:", Btree),nl,
    write("---------------------------\n"),
    bt_keys(Db_selector,Bt_selector,Key,Ref),
    write("\n",Key, " - ",Ref),
    fail.
списокБД(_).

bt_keys(Db_selector,Bt_selector,Key, Ref):-
    key_first(Db_selector,Bt_selector,_),
    bt_keysloop(Db_selector,Bt_selector,Key,Ref).

bt_keysloop(Db_selector,Bt_selector,Key,Ref):-
    key_current(Db_selector,Bt_selector,Key,Ref).

bt_keysloop(Db_selector,Bt_selector,Key,Ref):-
    key_next(Db_selector,Bt_selector,_),
bt_keysloop(Db_selector,Bt_selector,Key,Ref).

goal
db_open(мояБД,імяФайлу,у_файл),
списокБД(мояБД).
```

Для поновлення втраченої інформації у випадку аварії в системі можна вести в іншому файлі журнал змін. Якщо файл з БД пошкоджений, є можливість його реанімувати, використовуючи журнал змін та дублікат файлу БД, як наведено у фрагменті коду

```
domains
logdom=insert(relation,dbdom,ref);
replace(relation,dbdom,ref,dbdom);
erase(relation,ref,dbdom);

predicates
logdbchange(Logterm):-
chein_insertz(logdba,logchain,logdom,logterm,_),
db_flush(logdba).
```

У кожному відкритому (збереженому) дереві є покажчик на його вузли. При відкритті/оновленні дерева цей покажчик позиціонується на початку

дерева. Якщо покажчик вже позиціонований на останній ключ у B+ дереві, тоді виклик предиката *ключ_наступний* виконуватиме вихід покажчика за межі дерева. Кожного разу, коли покажчик виходить за дерево, *ключ_попередній* завершується невдало. Щоб гарантувати позиціонування покажчика у межах B+ дерева, можна скористатися наступними предикатами, за умови наявності у дереві ключів:

```
predicates

мійКлюч_наступний(db_selector, bt_selector, ref)
мійКлюч_попередній(db_selector, bt_selector, ref)
мійКлюч_пошук(db_selector, bt_selector, string, ref)

clauses
мійКлюч_наступний (Dba, bt_selector, ref):-
    мійКлюч_попередній(Dba, bt_selector, ref),
!.
мійКлюч_попередній (Dba, bt_selector, ref):-
    ключ_наступний(Dba, bt_selector, ref),
!.
мійКлюч_наступний (Dba, bt_selector, ref):-
    ключ_наступний (Dba, bt_selector, ref),
!.
мійКлюч_наступний (Dba, bt_selector, ref):-
    ключ_попередній(Dba, bt_selector, ref),
fail.
мійКлюч_пошук (Dba, bt_selector, Key, ref):-
    ключ_пошук(Dba, bt_selector, Key, ref),
!.
мійКлюч_пошук (Dba, bt_selector, _, ref):-
    ключ_поточний(Dba, bt_selector, _, ref),
!.
мійКлюч_пошук (Dba, bt_selector, _, ref):-
    ключ_останній(Dba, bt_selector, ref).
```

11.2.6. Зміна структури зовнішніх баз даних

Один із способів зміни структури бази даних полягає у написанні невеликої програми, яка копіює в процесі внесення змін стару базу даних у нову. Інший – в отриманні спочатку змісту бази даних у текстовому файлі, внесенні до нього будь-яких змін за допомогою текстового редактора і копіювання модифікованої БД до нового файлу.

Для передачі тексту, що міститься у базі даних, в текстовий файл, база даних повинна задовольняти наступним вимогам: кожен ланцюжок у базі даних моделює відношення; всі терми бази даних належать одному домену.

У цьому випадку B+ дерева в текстовий файл не вивантажуються, передбачається, відповідно до першої умови, що B+ дерева можуть генеруватися за відносинами. У прикладі всі терми належать домену *mydom*, під час практичної реалізації передбачуваного підходу mydom замінюється реальним ім'ям і відповідним оголошенням.

У наведеному фрагмент записується вміст бази даних у текстовий файл, відкритий за допомогою *outfile*. Кожний рядок текстового файлу містить терм і ім'я відповідного ланцюжка. Імена терма і ланцюжка об'єднуються в домені *chainterm*:

```
constants
ifdef os_unix
імяФайлу = "/user/local/PDCProlog/programs/register/register.bin"
elsedef
ifdef os_os2
імяФайлу = "c:\\vip74\\vpi\\programs\\register\\exe\\register.bin"
elsedef
імяФайлу = "c:\\Program Files\\Vip74\\vpi\\programs\\register\\exe\\register.bin"
enddef
enddef

domains
селектор_БД = мояБД
chainterm = chain (string, адреса)
file = outfile
адреса = місто(індекс, назва);
особа(імя, прізвище, вулиця, індекс, код)
вулиця, імя, прізвище, zipcode, індекс, назва, код = string

predicates
печать(chainterm)
підклБД(string,string)

clauses
друк (X):-
   write(X),nl.
відклБД (селектор_БД,З_Файлу):-
   db_open(мояБД,БД_ селектор,у_файл),
   openwrite(outfile, З_Файлу),
   writedevice(з_Файлу),
   db_chains(мояБД, Ланцюжок),
   chain_terms(мояБД, Ланцюжок, мояБД,Терм,_),
   wr(chain(Ланцюжок, Терм)),
   fail.
відклБД(_,_):-
   closefile(з_Файлу),
   db_close(мояБД).
```

```
goal
відклБД (імяФайлу,"register.txt").
```

За допомогою наведеної Пролог-програми стає можливою генерація текстового файлу, визиваючи предикат *chainterm*.

Якщо БД використовувати у мережевому програмному забезпеченні або на одній з багатозадачних платформ, Пролог представляє наступні засоби поділу файлів:

- два різних режими доступу та три різних режими поділу для оптимізації швидкості при відкритті збереженої БД;
- групування доступів до БД в транзакції для забезпечення цілісності;
- перевірочні предикати для розпізнавання поновлень БД іншими користувачами.

Для того, щоб отримати доступ до зовнішньої БД у загальному режимі, необхідно відкрити існуючий файл БД за допомогою предиката *db_open* з арністю 4, вказавши *accessMode* і *denyMode*.

З *AccessMode* файл буде відкритий як для читання, і будь-які спроби поновлення файлу приведе до помилки під час виконання. Якщо це *ReadWrite,* файл відкривається для читання і запису. *AccessMode* також може використовуватися із предикатом *db_begintransaction*.

```
domains
db_selector = mydba

goal
db_open(mydba,"share.dba",readwrite,denynone),
db_begintransaction(mydba,readwrite),
% тут виконується оновлення та доступ до загальної бази даних
db_endtransaction(mydba),
db_close(mydba).
```

З *denyMode* в *denynone* всі інші користувачі будуть мати можливість оновлювати і читати файл, якщо він *denywrite*, інші користувачі не зможуть відкрити файл в *AccessMode* = *ReadWrite*, але можливо оновити файл, відкривши у *AccessMode*=*ReadWrite*. Якщо *db_open* викликається з *denymode=DenyAll,* інші користувачі не зможуть отримати доступ до файлу.

Для основних конфліктних вимог цілісність БД та мінімальне блокування файлів повинні виконуватися одночасно.

Предикат *db_begintransaction* гарантує підтримку цілісності БД та виконує відповідне блокування файлу. При одночасному зверненні до файлу, тільки одному процесу дозволено обновляти файл. Якщо був викликаний *db_begintransaction*, то перед наступним викликом *db_begintransaction* повинен бути викликаний *db_endtransaction* для тієї ж БД, щоб виключити помилку під час виконання. *db_begintransaction* призначений відзначити початок транзакції бази даних і виконувати блокування та перенавантаження файлових дескрипторів, має наступний формат:

db_begintransaction(db_selector Dbase, accessmode AccessMode) (i,i)

db_endtransaction відзначає кінець транзакції за форматом:

db_endtransaction(db_selector Dbase) (i)

Для прискорення роботи з БД слід встановити *AccessMode* в *read DenyMode* в *denywrite*.

Для розділення файлів у Пролозі використовують наступні предикати, наведені в табл. 11.5.

Таблиця 11.5. Предикати розділення файлів

Предикати	Призначення
1	2
db_open (i,i,i,i)	Предикат призначений для відкриття БД у режимі розділення *db_open(Dbase, Name, AccessMode, DenyMode)* *Dbase* – селектор БД, *Name* – ім'я файлу, *AccessMode* – *read readwrite, DenyMode – denynone/denywrite/denyall*
db_begintransaction (i,i)	Предикат визначає початок транзакції *db_begintransaction(Dbase, AccessMode)*
db_endtransaction (i)	Предикат визначає кінець транзакції *db_endtransaction(Dbase)*
db_updated	Предикат відстежує зміни БД, якщо він викликаний з

(i)	середини транзакції (успішний), у якій зміни викликані іншим користувачем відбулися після останнього виклику *db_begintransaction* *db_updatd(Dbase)*
db_updated (i,i)	Предикат успішний у випадку, колиу B+ дереві відбулися зміни *db_updatd(Dbase, Btree_Sel)*
db_setretry (i,i,i)	Предикат запускає процес повторно через деякий час у випадку, коли файл заблоковано іншим процесом *db_* setretry*(Dbase, SleepPeriod, RetryCount)*

Навіть при правильному використанні предикатів розділення файлів, вони забезпечують низький рівень цілісності БД, що розділяється. Для запобігання блокування файлів БД, транзакції треба робити невеликими для забезпечення блокування файлу на максимально короткий час. До того ж, всі предикати, що використовуються для знаходження та доступу до деяких елементів БД повинні бути згрупованими всередині однієї транзакції.

Використовуючи *db_begintransaction* та *db_endtransaction* є можливість розробляти власні ефективні реалізації високорівневого блокування. Групування доступів до файлу з середини дозволяє системі Visual Prolog забезпечувати цілісність своїх таблиць дескрипторів. Але на високому рівні необхідна перевірка, щоб різні логічні обмеження, які накладаються на додаток, враховувались в мережі з багаточисленими користувачами.

База даних має серійний номер (6-и байтове ціле число), який реалізується та записується на диск під час кожного оновлення БД. Предикат *db_begintransaction* порівнює локальну копію серійного номера з копією на диску. Якщо серійний номер і його копія різні, дескриптори перезавантажуються. У масиві виконується блокування 256 користувачів. Коли користувач хоче отримати доступ до файлу, незаблокований простір вміщується в масив блокування та блокується на час виконання транзакції. Це дозволяє декільком користувачам одночасно отримати доступ до файлу.

Наступний приклад використовує майже всі предикати зовнішніх БД. Працюючи у оперативній пам’яті, програма виконує наступні дії: записує до БД 100 термів, зчитує їх, заміняє кожен другий терм, дублює кількість термів, видаляє кожен другий терм, перевіряє предикатом *ref_term* кожен терм, обчислює розмір БД. Далі програма копіює БД у файл і двічі виконує на диску ту ж послідовність дій відносно БД, вкінці обчислює загальний час на виконання дій.

```
domains
тип_кор=кор(string)
db_selector=бд_кор

predicates
запис_в _БД(integer)
читати_БД()
читати_Посил(ref)
обчислити_БД(integer)
вважати_Посил(ref, integer, integer)
замінити_БД()
замінити_Посил(ref)
подвоїти_БД()
подвоїти_Посил(ref)
половина_БД()
половина_Посил(ref)
разом()

clauses
запис_в _БД(0):-!.

запис_в _БД(N):-
chain_inserta(бд_кор,ланцюжок_кор, тип_кор),
кор("VIP-система"),_),N1=N-1, запис_в _БД(N1).

читати_БД:-
db_chains(БД_кор,Ланцюжок),
   chain_terms(БД_кор, Ланцюжок, Тип_кор,Терм,Посилання),
   write("\n Посилання = ", Посилання, "Терм = ", Терм), fail.
читати_БД:-
db_chains(БД_кор,Ланцюжок),
   chain_first(БД_кор, Ланцюжок, Посилання),
читати_Посил(Посилання),fail.
читати_БД.

читати_Посил(Посилання):-
   ref_term(БД_кор, Посилання, Терм),nl,write(Терм), fail.

читати_Посил(Посилання):-
chain_next(БД_кор, Посилання,Наст),!,
```

```
читати_Посил(Наст).

читати_Посил(_).

замінити_БД:-
chain_ first(БД_кор, ланцюжок_кор, Посилання),
замінити_Посил(Посилання).

замінити_Посил(Посилання):-
   term_replace(БД_кор, Тип_кор, Посилання, кор("VIP інструментарій")),
chain_next(БД_кор, Посилання,NN),
chain_next(БД_кор, NN, Наст),!, замінити_Посил(Наст).
замінити_Посил(_).

половина_БД:-
   hain_last(БД_кор, ланцюжок_кор, Посилання),
   половина_Посил(Посилання).

половина_Посил(Посилання):-
chain_prev(БД_кор, Посилання,РР),
chain_prev(БД_кор, РР,Попереднє),
term_delete(БД_кор, ланцюжок_кор, Посилання),половина_Посил(Попереднє).
половина_Посил(_).

подвоїти_БД:-
chain_ first(БД_кор, ланцюжок_кор, Посилання),
   подвоїти_Посил(Посилання).

подвоїти_Посил(Посилання):-
chain_next(БД_кор, Посилання,Наст),!,
chain_insertafter(БД_кор, ланцюжок_кор, Тип_кор, Посилання),
кор("Професійне керівництво"),_), подвоїти_Посил(Наст).
подвоїти_Посил(_).

обчислити_БД(N):-
chain_ first(БД_кор, ланцюжок_кор, Посилання).

вважати_Посил(Посилання, 1, N).
вважати_Посил(Посилання, N, N2):-
chain_next(БД_кор, Посилання,Наст),!,
   N1=N+1,
   write("\n Замінити кожен другий терм"),
   замінити_БД,
write("\n Подвоїти число термів: "),
подвоїти_БД,
write("\n Видалити кожен другий терм"),
половина_БД,
write("\n Перевірити кожен терм"),
читати_БД,
обчислити_БД(N),
write("\n У БД", N , "термів"),
db_statistics(БД_кор,ЧислоТермів,РозмПамяті,РозмБД,_),
```

```
write("\n Число термів=%, Розмір пам'яті=%, Розмір БД=%",
ЧислоТермів, РозмПамяті,РозмБД).

goal
write("\n\n\n\t Тест системи БД \n\t","----------\n\n"),time(Г1,Х1,С1,Д1),
db_create(БД_кор, "dd_dat",in_mamory).
write("\n\n Запис декількох термів у БД"), запис_в _БД(50),
читати_БД, разом,
write("\n\n Копіювати у файл"),db_copy(БД_кор, "dd_dat",in_file),
db_close(БД_кор),db_remove("dd_dat", in_mamory),
db_open(БД_кор, "dd_dat", in_file),
разом, db_close(БД_кор),
write("\n\n Відкриття файлу з БД"),
db_open(БД_кор, "dd_dat", in_file),
разом, db_close(БД_кор),
time(Г2,Х2,С2,Д2),
Час=(Д2-Д1)+100.0*(С2-С1)+60.0*(Х2-Х1)+60.0*(Г2-Г1),
Write("\n Час=",Час, "/100"),nl.
```

Запитання. Завдання

1. З чого складаються ланцюжки зовнішньої БД?
2. За допомою чого можна отримати доступ до ланцюжків БД?
3. До якого домену відносять покажчики БД?
4. За допомогою чого ідентифікується індивідуальність БД?
5. До якого домена відносять селектори БД?
6. Місце збереження зовнішньої БД?
7. Місце і призначення B+ дерева у БД?
8. За допомогою чого позначаються B+ дерева?
9. Структура B+ дерева.
10. Чим визначається число ключів, що згруповані в одному візлі B+ дерева?
11. Яким чином виконується поділ файлів?

Тест для самоконтролю знань з розділу 11[Ғ]

1. Як оголошується внутрішня база фактів?
   а) предикати, що описують внутрішню базу фактів, оголошують у розділі програми *facts;*
   б) предикати, що описують внутрішню базу фактів, оголошують у розділі програми *predicates;*

[Ғ] Усі запитання мають один правильний варіант відповіді.

в) предикати, що описують внутрішню базу фактів, оголошують у розділі програми *predicates;*

г) предикати, що описують внутрішню базу фактів, оголошують у розділі програми *constants.*

2. Внутрішня база фактів може поповнюватися …
   а) тільки фактами*;*
   б) фактами і правилами*;*
   в) тільки правилами.
3. Які вимоги до імен предикатів внутрішньої бази фактів?
   а) не можна застосовувати однакові імена предикатів у двох різних розділах facts;
   б) не можна застосовувати однакові імена предикатів у двох різних розділах facts та у розділах facts і predicates*;*
   в) не можна застосовувати однакові імена предикатів у розділі facts і predicates.
4. Значення ключового слова *nocopy* при оголошенні бази фактів?
   а) визначає глобальну базу фактів;
   б) визначає, що база фактів може містити будь-яке число фактів для предиката бази фактів;
   в) визначає, що база фактів завжди містить один і тільки один факт для предиката бази фактів;
   г) оголошує, що дані не будуть скопійовані, а змінні будуть посилатися безпосередньо на дані факту, що зберігаються у наборі.
5. Які особливості однократних фактів внутрішньої бази фактів?
   а) виклик однократного факту завжди неуспішний;
   б) не можуть бути видаленими, у випадку такої спроби компілятор генерує помилку.
6. При виконанні програми факти з внутрішньої бази фактів …
   а) компілюються у двійковий код;
   б) як стандартні предикати викликаються з таблиці, яку можна легко змінити.
7. Обмеження при використанні внутрішньої бази фактів …
   а) проблема використання оперативної пам’яті комп’ютера при завантажені ВБФ;
   б) неможливість збереження фактів у текстовому файлі програми мовою Пролог.
8. Призначення B+ дерев?
   а) використовують у внутрішній БФ для швидкого доступу до даних;
   б) використовують у зовнішній БД для швидкого доступу до даних.
9. Особливості запису імен стандартних предикатів управління базами даних?
   а) перша частина імені ідентифікує вхідні дані, наступна – указує на вихідні;

б) вимоги аналогічні вимогам до звичайних предикатів.

10. З чого служить числовий покажчик у ланцюжку зовнішньої БД?
    а) для вибірки, переміщення або заміни терма;
    б) для сортування чи пошуку терма.
11. При створенні або відкритті БД її можна помістити …
    а) тільки в оперативну пам’ять;
    б) тільки у файл;
    в) у файл або в оперативну пам’ять.
12. Які обмеження на змішування доменів у зовнішній БД?
    а) оголошувати всі терми БД як альтернативи одного домена;
    б) немає обмежень на змішування доменів.
13. Яка відповідь відноситься до B+ дерева?
    а) шляхи для кожного ключа у «листях» дерева мають одну й ту ж довжину;
    б) у кожному вузлі запам’ятовується більше одного ключового рядка.
14. У деревах B+ ключі групуються …
    а) у вузли;
    б) у сторінки.
15. У Visual Prolog B+ дерева зберігаються у зовнішніх БД та забезпечують …
    а) індексування БД, де кожен вхід у B+ дерево відповідає ключовому рядку та, зв’язаному з ним, покажчику бази даних;
    б) індексування БД, де кожен вхід у B+ дерево відповідає сторінці та, зв’язаному з нею, терму.

РОЗДІЛ 12. ПРОГРАМУВАННЯ НА СИСТЕМНОМУ РІВНІ У VISUAL PROLOG

Visual Prolog містить стандартні предикати, за допомогою яких можна виконувати звернення до операційної системи.

Також дана мова програмування містить предикати побітової обробки чисел. За допомогою ряду предикатів виконується підтримка низькорівневого режиму роботи з пам'яттю

Далі перелічені предикати, які використовує Visual Prolog для програмування на системному рівні (табл. 12.1).

Таблиця 12.1. Предикати доступу до операційної системи

Предикат	Призначення
1	2
system (i)	Предикат призначений для для отримання доступу до операційної системи. Якщо аргумент є пустим рядком, то командний інтерпретатор завантажується в інтерактивному режимі. System *(«command»)*
system (I,I,o)	Предикат призначений для для отримання доступу до операційної стреми. *System(CommandString, Resetvideo, ErrorLevel),* де Resetvideo *ErrorLevel* зв'язується із значенням рівня помилки ОС: *Resetvideo* =1 – повертає у попередній стан; *Resetvideo* =0 – не повертає (в екранному режимі).
Envsymdol (I,i)	Предикати призначені для пошуку змінних середовища у таблиці операційної системи – неуспішний, якщо задана змінна середовища не визначена. *Envsymbol(EnvSymb, Value)*
Comline (o)	Предикат прочитує параметри командного рядка, що використовувався під час виклику програми *comline (CoiranandLine)*

Продовження таблиці 12.1

1	2

date (i,i,i) date (o,o,o) date(o,o,o,o)	*date(Year, Month, Day)* обробляється аналогічно time. *Date(Year, Month, Day, WeekDay)* де *WeekDay* – номер дня тижня.
time (I,i,i,i) time (o,o,o,o)	У випадку, коли при виклику предиката всі змінні зв'язані, предикат переопреділяє системний годинник, інакше – система їх зв'язує із значенням, отриманим з системного годинника *time(Hours, Minutes, Seconds, Hundredths)*
syspath (o,o)	Використовується для надання програмам можливості завантаження файлів з їх домашнього каталога *syspath(HomeDir,ExeName)*

Таблиця 12.2. Сервіси часу

Предикат	Призначення
1	2
sleep (i)	Предикат припиняє виконання програми на заданий час *sleep(CentiSecs),* де *CentiSecs* – час, вимірюваний у сотих частках секунди, на яке програма буде припинена.
Mrktime (i,o)	Предикат повертає відмітку часу *marktime(CentiSecs,Ticket)* *CentiSecs* – необхідний час, протягом якого *Ticket* має продовжуватися. *Ticket* призначена тільки для подальшого використовування предикатами *timeout* і *difftime*.
dfftime (i,i,o)	Предикат повертає різницю між першим і другим тимчасовими маркерами у форматі дійсного числа *difftime(real,real,real)*

Продовження таблиці 12.2

1	2
Tmeout (i)	Предикат перевіряє на закінчення відмітку часу, повернену marktime *timeout(Ticket)*

sound (i,i)	Предикат генерує звук апаратного спікера. *Duration* показує час у сотих частках секунди. *Sound(Duration,Frequency)*
beep	Предикат еквівалентний *sound*
osversion (o)	Предикат повертає версію поточної ОС *osversion(VerString)*
diskspace (i,o)	Предикат повертає в 395ез знакових395 довгому цілому доступний простір на диску. *Where* –позначає символ диска *diskspace(Where,Space)*
storage (o,o,o)	Предикат повертає інформацію про три області пам'яті, що використовуються системою для виконуваних програм у вигляді 395ез знакових довгих цілих *storage(StackSize, HeapSize, TrailSize)* *storage/11* повертає докладнішу інформацію. *Storage* – друкує у спеціальному вікні, скільки пам'яті використовують різні частини системи управління пам'яттю Visual Prolog і число точок повернення.

Таблиця 12.3. Предикати для бітових операцій

Предикат	Призначення
1	2
bitnot (i,o)	Предикат виконує функції побітового логічного «НІ» bitnot (X, Z) *Оператор X z* *Bitnot 1 0* *0 1*

Продовження таблиці 12.3

1	2
bitand (i,i,o) bitand(X, У, Z)	Предикат побітно виконує логічну операцію «І» *Операто X У z* *р* *Bitand 1 1 1* *1 0 0* *0 1 0* *0 0 0*

bitor (i,i,o)	Предикат виконує побітове логічне «*АБО*» *bitor(X, У, Z)* *Оператор X У z* *bitor 1 1 1* *1 0 1* *0 1 1* *0 0 0*
bitxor (i,i,o)	Предикат виконує те, що побітове логічне «*виключає АБО*»: *bitxor(X, У, Z)* *Операто X У z* *p* *bitxor 1 1 0* *1 0 1* *0 1 1* *0 0 0*
bitleft (i,i,o)	Предикат виконує побітовий зсув вліво *bitleft(X, N, У)*
bitright (i,i,o)	Предикат виконує зсув вправо *bitright(X, N, У)*

Таблиця 12.4. Прямий (низькорівневий) доступ до пам'яті

Предикат	Призначення
1	2
ptr_dword (o,i,i) ptr_dword (i,o,o)	Предикат повертає внутрішню адресу *stringvar* або створює рядок *stringVar* за вказаною адресою в пам'яті *ptr_dword(StringVar, Segment, Offset)* Якщо *stringVar* зв'язана, то предикат *ptr_dword* повертає номер сегмента (*Segment*) і зсув для рядка *stringVar*.

	Якщо зв'язані *Segment* і *Offset*, предикат *ptr_dword* пов'язує *stringVar* з рядком, який зберігається у вказаному місці. На 32-бітових платформах сегмент ігнорується. Предикат *ptr_dword* значною мірою був витиснений функцією *cast*. Символьним рядком у Visual Prolog є послідовність символів у кодах ASCII, що закінчується нульовим значенням. Рядки, що містять нульові байти не можуть заноситися до бази даних або вилучатися з неї.
memByte (i,i,i) memByte (i,i,o) memByte (i,i) memByte (i,o)	Предикат використовується для доступу до *byte* *memByte(Segment, Offset, Byte)* *memByte (StringVar, Byte)*
memWord (i,i,i) memWord (i,i,o) memWord(i,i) memWord (i,o)	Предикат використовується для доступу до *Word* *memWord(Segment, Offset, Word)* *memWord(StringVar, Word)*
memDword (i,i,i) memDword (i,i,o) memDword (i,i) memDword (i,o)	Предикат використовується для доступу до *Dword* *memDword(Segment, Offset, DWord)* *memDword(StringVar, DWord)*

Приклади використовування предикату *system(i)*.

```
system("\"C:\\VIP Support Tests\\Run system.exe\"")
```

Приклади використовування предикату *system(i,i,o)*.

```
system("\"C:\\VIP Support Tests\\Run system.exe\"", 1, _Exit)
```

Приклади використовування предикату *envsymdol (i,i)*

```
SET SYSDIR=C:\FOOL
/*...*/
envsymbol("SYSDIR", SysDir),
/*...*/
```

Приклади використовування предикату *time (o,o,o,o)*

```
goal
time(H,M,S,T).
```

```
/*H=11, M=30, S=13, T=49
1 Solution*/
```

Приклади використовування предикату *date (o,o,o)*

```
goal

date(Y,M,D)
/*Y=1991, M=12, D=17
1 Solution*/
```

Приклади використовування предикату *comline (o)*

```
predicates
  extend(string,string)
  getfilename(string,string)

  clauses
  extend(S,S):- concat(_,".pro",S),!.
  extend(S,S1):- concat(S,".pro",S1).
  % Виберіть ім'я з каталогу, як аргумент
  getfilename("",Fname):- dir("","*.pro",Fname),!.
  % додавання розширення. PRO, якщо не вказане розширення
  getfilename(X,X1):- extend(X,X1).

  goal
  /*зчитування параметрів командного рядка, що використовувався під час
    виклику програми*/
  comline(X),
  getfilename(X,X1),% отримати ім'я файлу
  file_str(X1,S),
  textmode(Рядки,Стовбці),CC=Cols-1,
  makewindow(1,23,0,"editor",0,0,Рядки,CC), % створити
  edit(S,S1,"","","",0,"",R), % редагувати
  removewindow,% видалити
  R><1,!, % Залишається, якщо не натиснута [ESC]
file_str(X1,S1).
```

Приклади використовування предикату *syspath (o,o)*

```
goal
syspath(P,N).
/*P=D:\Prolog7\PSYS\, N=PROLOG7.EXE
1 Solution*/
```

Приклади використовування предикату *sleep (i)*

```
goal
sleep(100). % Режим «Sleep» протягом 1 секунди
```

Приклади використовування предикату *sound(i,i)*

```
predicates
мелодія(char)

clauses
мелодія ('e'):- !, sound(100, 165).
```

```
мелодія ('a'):- !, sound(100, 220).
мелодія ('d'):- !, sound(100, 294).
мелодія ('g'):- !, sound(100, 392).
мелодія ('b'):- !, sound(100, 494).
мелодія ('h'):- !, sound(100, 660).

мелодія (_):-
clearwindow,   % очистити
write("Потрібне налаштування!\n\n",
"Натисніть будь-яку клавішу для продовження"),
readchar(_), clearwindow,
fail.                %предикат readchar зчитує символ

goal
makewindow(1, 7, 7, "тюнер гітари",0,0,25,80),
write("Це програма, яка допоможе Вам налаштувати гітару \n\n"),
write("Будь ласка, виберіть, що Ви хочете налаштувати \n",
      "(e, a, d, g, b, або h):"),
readchar(Мелодія),
upper_lower(Мелодія, Мелодія1),
мелодія(Мелодія1).
```

Приклади використовування предикату *beep*

```
predicates
input_integer(integer)

clauses
input_integer(X):-
write("Написати ціле: "), readint(X), !.
input_integer(X):- beep, input_integer(X).

goal
input_integer(X), write(X).
```

Приклади використовування предикату *diskspace (i,o)*

```
goal
infdef os_unix
disk(C), % Отримати робочий поточний каталог
frontchar(C, Диск, _),
diskspace(Диск, Розмір),

elsedef
diskspace("",Розмір),

enddef
write(Розмір," байти вільного об'єму.\n").
```

Приклади використовування предикату *bitnot (i,o)*

```
goal
bitnot(0,X).    %32-bit system
%X=65535
```

Приклади використовування предикату *bitxor (i,i,o)*

```
goal
bitxor(1,3,X).
%X=2.
```

Запитання. Завдання

1. Які предикати виконують доступ до операційної системи?
2. Які предикати виконують побітові логічні операції?
3. Які предикати виконують підтримку прямого доступу до пам’яті?
4. Читання аргументів командного рядка.
5. Побітові операції переміщення.
6. Повернення стартового каталога і імені виконуючого файлу.
7. Доступ до вбудованого календаря.
8. Повернення об’єму вільного місця на диску?
9. Можливість виконання зовнішньої програми.
10. Доступ до таблиці змінних середовища ОС?

СПИСОК ЛІТЕРАТУРИ

1. Адаменко А.Н. Логическое программирование и Visual Prolog. / А.Н. Адаменко, А.М. Кучуков. – СПб. : БХВ-Петербург, 2003. – 992 с.
2. Стерлинг Л. Искусство программирования на языке Пролог. / Л. Стерлинг, Э. Шапиро. – Пер. с англ. М. : Мир, 1990. – 580 с.
3. Братко И. Программирование на языке Пролог для искусственного интеллекта. / И. Братко. – Пер. с англ. М. : Мир, 1990. – 560 с.
4. Братко И. Алгоритмы искусственного интеллекта на языке PROLOG. / И. Братко. – М. : Вильямс, 2004. – 640 с.
5. Чанышев О.Г. Логическое программирование и Visual Prolog. / О.Г. Чанышев. – СПб.: БХВ-Петербург, 2003.
6. Трушевський В. М. Технології та мови програмування для штучного інтелекту. Частина 1: Основи програмування мовою Prolog. / В.М. Трушевський. – Львів : Львівська політехніка, 2006. – 120 с.
7. Різник О.Я. Логічне програмування: навчальний посібник. / О.Я. Різник. – Львів : Львівська політехніка, 2008. – 332 с.
8. Тарасенко О.П. Практикум з логічного програмування. / О.П. Тарасенко, С.Г. Волков. – Харьков, 2009. – 92 с.
9. Солдатова О.П. Логическое программирование на языке Visual Prolog. / О.П. Солдатова, И.В. Лезина. – Самара : СНЦ РАН, 2010 – 81 с.
10. Цуканова Н.И. Логическое программирование на языке Visual Prolog. / Н.И. Цуканова, Т.А. Дмитриева. – М. : Горячая линия-Телеком, 2008. – 144 с.
11. Михайлов Д.В. Функциональное и логическое программирование. Ч.2 Логическое программирование : Лабораторный практикум. / Д.В. Михайлов, Г.М. Емельянов. – Великий Новгород : НовГУ им. Ярослава Мудрого, 2007. – 88 с.

12. Будилов В.Н. Учебно-методический комплекс по дисциплине «Интеллектуальные системы». / В. Н. Будилов, В. И. Воловач. – Тольятти : ПВГУС, 2012. – 196 с.
13. Гаврилова Т.А. Базы знаний интеллектуальных систем. / Т.А. Гаврилова, В.Ф. Хорошевских. – Питер : СПб, 2000, – 450 с.
14. Джексон П. Введение в экспертные системы. / П. Джексон. – М. : Вильямс, – 2001.
15. Пономарев В.Ф. Математическая логика. Учебное пособие. / В.Ф. Пономарев. – Калининград. : КГТУ, 2005. – 201 с.
16. Visual Prolog 7.4 Language Reference Prolog Development Center [Електронний ресурс]. – Режим доступу: www.visual-prolog.com
17. Eduardo Costa. Visual Prolog 7.0 for Tyros. [Електронний ресурс]. – Режим доступу: www.ProgBook.net

Навчальне видання

Шумейко *Олександр Олексійович*
Кнуренко *Валентина Миколаївна*

VISUAL PROLOG.
ОПАНУЙ НА ПРИКЛАДАХ

Навчальний посібник

Видання друкується в авторській редакції

Відповідальний редактор *Біла К.О.*
Оригінал-макет *Косолапов О.В.*
Технічний редактор *Капуш О.Є.*

Підп. до друку 10.12.14. Формат 60x84$^{1}/_{16}$.
Ум. друк. арк. 22,12. Тираж 300 пр. Зам. № 1111-10.

Видавець та виготовлювач СПД Біла К. О.
Свідоцтво про внесення до Державного реєстру
Суб'єктів видавничої справи ДК № 3618 від 06.11.2009

Надруковано на поліграфічній базі видавця Білої К. О.
Поштова адреса: Україна, 49087, м. Дніпропетровськ,
п/в 87, а/с 4402

тел. +38 (067) 972-90-71

www.confcontact.com e-mail:
conf@confcontact.com