Програмування інтелектуальних інформаційних систем

3 курс, осінь 2021

- Доц. Баклан І.В.
- Email: iaa@ukr.net
- Web: baklaniv.at.ua

Лекція 8

Логічний висновок засобами Ліспу. Перехід до Пролога

У цій лекції ми покажемо, як написати програму, що здійснює логічні «умовиводи», засновані на наборі правил «якщо-то». Це класичний приклад, який не тільки часто зустрічається в підручниках, а й відображає початкову концепцію Лиспа як мови «символьних обчислень». Багато ранніх програми на Ліспі мають риси тієї, яка описана нижче.
Тим самим прокладемо місток до ще однієї мови штучного інтелекту — Пролог.

У даній програмі ми збираємося подавати інформацію в знайомій нам формі: списком з предиката і його аргументів. Уявімо факт батьківства Дональда по відношенню до Ненсі таким чином:

```
(Parent donald nancy)
```

Крім фактів наша програма буде використовувати правила, які повідомляють, що може бути виведено з наявних фактів.

Ми будемо представляти ці правила так:

```
(<- заголовок тіло)
```

де **заголовок** відповідає слідству, а **тіло** - умові. Змінні всередині них ми будемо представляти символами, що починаються зі знака питання. Таким чином, наступне правило:

```
(<- (child ?x ?y) (parent ?y ?x))
```

повідомляє, що якщо **y** є батьком **x**, то **x** є дитиною **y**, або, кажучи більш точно, ми можемо довести будь-який факт виду **(child x y)** через доказ **(parent y x)**.

Тіло (умовна частина правила) може бути складовим виразом, що включає логічні оператори **and**, **or**, **not**.

Так, правило, згідно з яким, якщо **x** є чоловіком і батьком **y**, то **x** - батько **y**, виглядає наступним чином:

```
(<- (father ?X ?Y) (and (parent ?X ?Y)
(male ?X)))
```

Одні правила можуть спиратися на інші. Наприклад, правило, що визначає, чи є **x** дочкою **y**, спирається на вже певне правило батьківства:

```
(<- (daughter ?X ?Y) (and (child ?X ?Y)
(female ?X)))
```

При доказі виразів може застосовуватися будь-яку кількість правил, необхідних для досягнення твердого грунту фактів. Цей процес іноді називають *зворотнім ланцюжком логічного висновку (backward chaining)*. Зворотною тому, що висновок спершу розглядає частину-наслідок, щоб побачити, чи буде правило корисним, перш ніж продовжувати доказ частини-умови. А ланцюжком він називається тому, що правила залежать один від одного, формуючи ланцюжок (скоріше навіть дерево) правил, яка веде від того, що ми хочемо довести, до того, що ми вже знаємо.

Щоб написати нашу програму для зворотного ланцюжка логічного висновку, нам знадобиться функція, що виконує *зіставлення зі зразком (pattern-matching)*. Ця функція повинна порівнювати два списки, які, можливо, містять змінні, і перевіряти, чи є така комбінація значень змінних, при якій списки стали б рівними.

Наприклад, якщо **?x** і **?y** - змінні, то два списки:

```
(p ?x ?y c ?x)
(p a b c a)
```

співставляються, якщо **?x = a** і **?y = b**, а списки

```
(p ?x b ?y a)
(p ?y b c a)
```

співставляються при **?x =?y = c**.

Нижче показана функція **match**, що співставляє два дерева.

```
[1]> (defun match (x y &optional binds)
(cond
((eql x y) (values binds t))
((assoc x binds) (match (binding x binds)
y binds))
((assoc y binds) (match x (binding y
binds) binds))
((var? x) (values (cons (cons x y) binds)
t))
((var? y) (values (cons (cons y x) binds)
t))
(t
```

```
 (when (and (consp x) (consp y))
 (multiple-value-bind (b2 yes)
 (match (car x) (car y) binds)
 (and yes (match (cdr x) (cdr y) b2)))))))
MATCH
```

```
[2]> (defun var? (x)
(and (symbolp x)
(eql (char (symbol-name x) 0) #\?)))
VAR?
[3]> (defun binding (x binds)
(let ((b (assoc x binds)))
(if b
(or (binding (cdr b) binds)
(cdr b)))))
BINDING
```

Якщо дерева можуть збігатися, то вона повертає асоціативний список, який показує, яким чином вони збігаються:

```
> (match '(p a b c a) '(p ?x ?y c ?x))
((?Y . B) (?X . A))
T
> (match '(p ?x b ?y a) '(p ?y b c a))
((?Y . C) (?X . ?Y))
T
> (match '(a b c) '(a a a)
NIL
```

У міру поелементного порівняння **`match`** накопичує привласнення змінним значень, які ми називаємо *зв'язками (bindings)*, в змінної **`binds`**. Якщо зіставлення завершилося успішно, то **`match`** повертає згенеровані зв'язку, в іншому випадку повертає **`nil`**.
Так як не всі успішні зіставлення генерують хоч якісь зв'язки, **`match`**, за тим же принципом, що і **`gethash`**, повертає друге значення, яке вказує на успішність зіставлення.

Коли **match** повертає **nil** і **t**, як сталося вище, це означає, що має місце успішне зіставлення, не створила зв'язків для змінних. Алгоритм дії **match** можна описати таким чином:

1. Якщо **x** і **y** рівні з точки зору **eql**, то вони зіставляються; інакше:
2. Якщо **x** - це змінна, вже асоційована зі значенням, то вона зіставляється з **y**, якщо значення збігаються; інакше:
3. Якщо **y** - це змінна, вже асоційована зі значенням, то вона зіставляється з **x**, якщо значення збігаються; інакше:

4. Якщо змінна **x** не асоційована зі значенням, то з нею зв'язується поточне значення; інакше:
5. Якщо змінна y не асоційована зі значенням, то з нею зв'язується поточне значення; інакше:
6. Два значення зіставляються, якщо вони обидва - **cons**-комірки, їх **car**-елементи зіставляються, а **cdr** зіставляються з урахуванням отриманих зв'язків.

Ось два приклади, які демонструють по порядку всі шість випадків:

```
> (match '(p ?v b ?x d (?z ?z))
         '(p a ?w c ?y ( e e))
         '((?v . a) (?w . b)))
((?Z . E)(?Y . D)(?X . C)(?V . A)(?W . B))
T
```

Щоб знайти асоційоване значення (якщо воно є), **`match`** викликає **`binding`**. Ця функція повинна бути рекурсивної, так як в результаті зіставлення можуть вийти пари, в яких змінна пов'язана зі значенням побічно, наприклад **`?x`** може бути пов'язаний з **`a`** через список, що містить **`(?x . ?y)`** і **`(?y . a)`**.

```
> (match '(?x a) '(?y ?y))
((?Y . A) (?X . ?Y))
T
```

Зіставивши **`?x`** з **`?y`**, а **`?y`** з **`a`**, ми бачимо непрямий зв'язок змінної **`?x`** зі значенням.

Тепер, коли введені основні концепції зіставлення, ми можемо перейти безпосередньо до призначення нашої програми: якщо ми маємо вираз, ймовірно, що містить змінні, то виходячи з наявних фактів і правил ми можемо знайти всі асоціації, що роблять цей вислів істинним. Наприклад, якщо у нас є тільки той факт, що

```
(parent donald nancy)
```

і ми просимо програму довести

```
(parent ?x ?y)
```

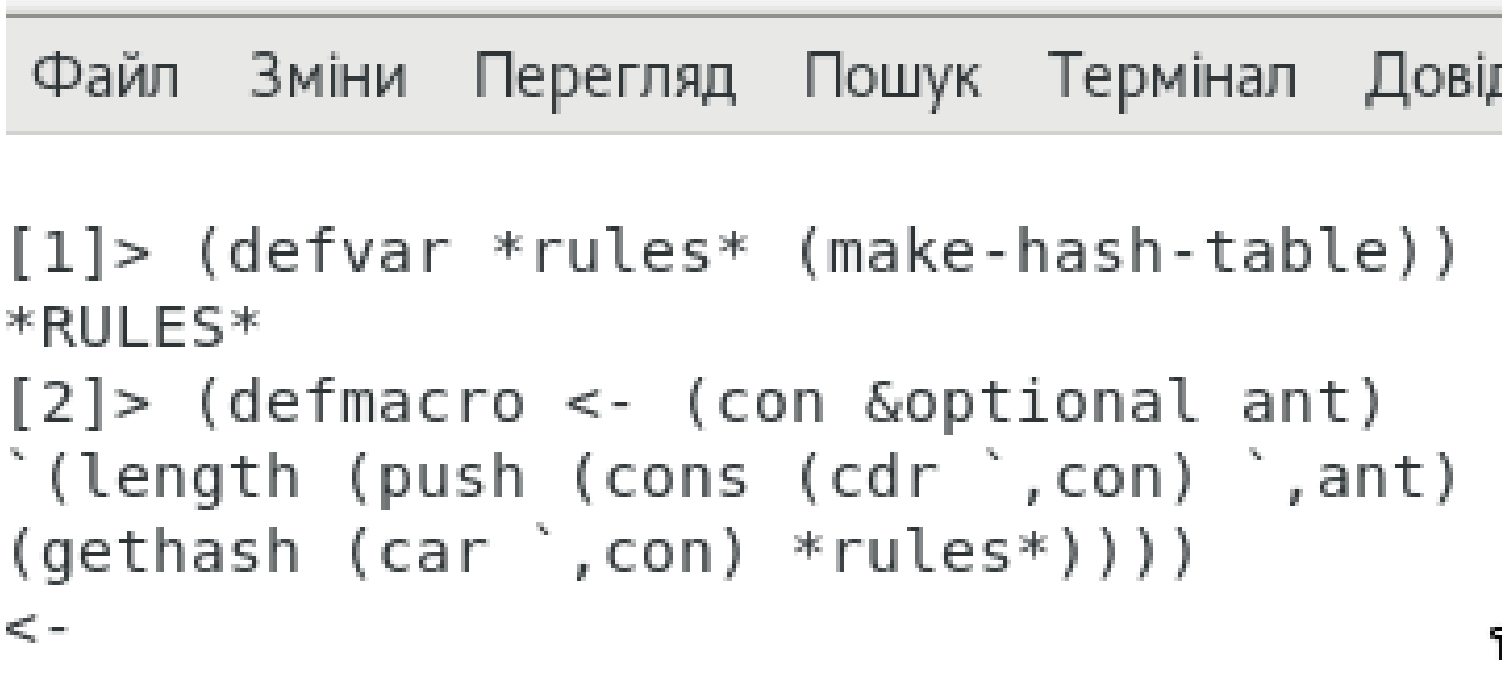
то вона поверне щось, схоже на

```
(((?x . donald) (?y . nancy)))
```

Це означає, що наше вираз може бути істинним лише в одному випадку: **?x** - це **donald**, а **?y** - **nancy**.

Оскільки у нас є функція зіставлення, можна стверджувати, що ми на правильному шляху. Тепер нам потрібно навчитися визначати правила. Відповідний код наведено нижче. Правила містяться в хеш-таблиці `*rules*` відповідно до предикатами в заголовках. Це вводить обмеження, що змінні не можуть перебувати на місці предикатів. Від нього можна позбутися, якщо зберігати такі правила в окремому списку, але тоді для доказу чого-небудь нам довелося б зіставляти один одному кожне правило з цього списку.

```
[1]> (defvar *rules* (make-hash-table))
*RULES*
[2]> (defmacro <- (con &optional ant)
`(length (push (cons (cdr `,con) `,ant)
(gethash (car `,con) *rules*))))
<-
```



Ми будемо використовувати макрос **<-** для визначення і правил, і фактів. Факт представляється у вигляді правила, що містить тільки заголовок. Це відповідає нашому уявленню про правила. Правило говорить, що для доказу заголовка необхідно довести тіло, а раз тіла немає, то і доводити нічого. Ось два вже знайомих нам приклади:

```
> (<- (parent donald nancy))
1
> (<- (child ?x ?y) (parent ?y ?x))
1
```

Виклик **`<-`** повертає номер нового правила для заданого предиката; обертання **`length навколо push`** дозволяє уникнути великого обсягу інформації, що виводиться в **`toplevel`**.

Нижче міститься велика частина коду, потрібного нам для виведення.

```
(defun prove (expr &optional binds)
(case (car expr)
(and (prove-and (reverse (cdr expr))
binds))
(or (prove-or (cdr expr) binds))
(not (prove-not (cadr expr) binds))
(t (prove-simple (car expr) (cdr expr)
binds))))
(defun prove-simple (pred args binds)
(mapcan #'(lambda (r)
(multiple-value-bind (b2 yes)
(match args (car r) binds)
```

```
(when yes
(if (cdr r)
(prove (cdr r) b2)
(list b2)))))
(mapcar #'change-vars
(gethash pred *rules*))))
(defun change-vars (r)
(sublis  (mapcar  #'(lambda  (v)  (cons  v
(gensym "?")))
(vars-in r)) r))
(defun vars-in (expr)
(if (atom expr)
(if (var? expr) (list expr))
(union (vars-in (car expr))
(vars-in (cdr expr)))))
```

Функція prove є віссю, навколо якої обертається весь логічний висновок. Вона приймає вираз і необов'язковий набір зв'язків. Якщо вираз не містить логічних операторів, то викликається **`prove-simple`**. Саме тут відбувається сам зворотний висновок. Ця функція шукає все правила з істинним предикатом і намагається зіставити заголовок кожного з них з фактом, який ми хочемо довести. Потім для кожного співпала заголовка доводиться його тіло з урахуванням нових зв'язків, створених **`match`**. Виклики **`prove`** повертають списки зв'язків, які потім збираються **`mapcan`**:

```
> (prove-simple 'parent '(donald nancy)
nil)
(NIL)
> (prove-simple 'child '(?x ?y) nil)
(((#:?6 . NANCY) (#:?5 . DONALD) (?Y . #:?
5) (?X . #:?6)))
```

Обидва отриманих значення підтверджують, що є лише один шлях докази. (Якщо довести твердження не вийшло, повертається `nil`.) Перший приклад був виконаний без створення будь-яких зв'язків, в той час як у другому прикладі змінні `?x` і `?y` були пов'язані (опосередковано) з `nancy` і `donald`.

Між іншим, ми бачимо тут хороший приклад висловленої раніше в лекціях ідеї: оскільки наша програма написана в функціональному стилі, ми можемо тестувати кожну функцію окремо.

Тепер пара слів про те, навіщо потрібен **gensym**. Раз ми збираємося використовувати правила, що містять змінні, то нам потрібно уникати наявності двох правил з однієї і тієї ж змінної. Розглянемо два правила:

```
(<- (child ?x ?y) (parent ?y ?x))
(<- (daughter ?y ?x) (and (child ?y ?x) (female ?y)))
```

У них стверджується, що для будь-якого **x** і **y** 1) **x** є дитиною **y**, якщо **y** є батьком **x**; 2) **y** є донькою **x**, якщо **y** є дитиною **x** і **y** - жінка. Зв'язок між змінними в рамках одного правила має велике значення, а ось факт, що два правила використовують однакові імена для визначення змінних, абсолютно випадковий.

Якщо ми скористаємося цими правилами в тому вигляді, в якому тільки що їх записали, то вони не будуть працювати. Якщо ми спробуємо довести, що **a** - дочка **b**, то зіставлення з заголовком другого правила дасть зв'язку **?y = a** і **?x = b**. Маючи такі зв'язки, ми не зможемо скористатися першим правилом:

```
> (match '(child ?y ?x)
'(child ?x ?y)
'((?y . a) (?x . b)))
NIL
```

Щоб переконатися у тому, що змінні будуть діяти лише в межах одного правила, ми замінюємо всі змінні всередині правила на **`gensym`**. Для цієї мети визначена функція **`change-vars`**. Так ми набуваємо за щиту від збігів з іншими змінними в визначеннях інших правил. Але, оскільки правила можуть застосовуватися рекурсивно, нам також потрібен захист при зіставленні з цим же правилом. Для цього **`change-vars`** повинна викликатися не тільки при визначенні правил, а й при кожному їх використанні.

Тепер залишається лише визначити функції для доказу складових тверджень. Відповідний код наведено нижче. Обробка виразів `or` або `not` гранично проста. У першому випадку ми збираємо всі зв'язки, отримані з кожного виразу в `or`. У другому випадку ми просто повертаємо наші зв'язки, якщо вираз всередині `not` повернуло `nil`.

```
(defun prove-and (clauses binds)
 (if (null clauses)
  (list binds)
  (mapcan #'(lambda (b)
  (prove (car clauses) b))
  (prove-and (cdr clauses) binds))))
(defun prove-or (clauses binds)
(mapcan #'(lambda (c) (prove c binds))
         clauses))
(defun prove-not (clause binds)
 (unless (prove clause binds)
  (list binds)))
```

Функція **`prove-and`** лише трохи складніше. Вона працює як фільтр, доводячи перший вираз для кожного з наборів зв'язків, отриманих з інших виразів. З цієї причини вираження всередині **`and`** розглядаються в зворотному порядку. Перевертання результату **`prove-and`** компенсує це явище (так як результуючий список виразів формується функцією в зворотному порядку, в кінці її виконання цей список потрібно розгорнути).

Тепер у нас є робоча програма, але вона не дуже зручна для кінцевого користувача. Розбирати списки зв'язків, що повертаються prove, досить складно, але ж з ускладненням виразів вони будуть тільки зростати. Цю проблему вирішує макрос **with-answer**, зображений нижче.

```
(defmacro with-answer (query &body body)
 (let ((binds (gensym)))
  `(dolist (,binds (prove ',query))
  (let ,(mapcar #'(lambda (v)
   `(,v (binding ',v ,binds)))
  (vars-in query))
   ,@body))))
```

Він приймає запит (НЕ обчислений) і обчислює своє тіло для кожного набору зв'язків, отриманих при обробці запиту, пов'язуючи кожну змінну запиту зі значенням, яке вона має в текущемекземпляре зв'язків:

```
> (with-answer (parent ?x ?y)
(format t "~A is the parent of ~A.~%" ?x ?y))
DONALD is the parent of NANCY.
NIL
```

Цей макрос розшифровує отримані зв'язку і надає зручний спосіб використання prove в наших програмах. Нижче показаний результат його розкриття, а ще нижче демонструються деякі приклади використання цього макросу.

```
(with-answer (p ?x ?y)
(f ?x ?y))
;;; розкривається до:
(dolist (#:g1 (prove '(p ?x ?y)))
(let ((?x (binding '?x #:g1))
(?y (binding '?y #:g1)))
(f ?x ?y)))
```

Якщо ми виконаємо **(clrhash *rules*)** і потім визначимо такі правила і факти:

```
(←  (parent donald nancy))
(←  (parent donald debbie))
(←  (male donald))
(←  (father ?x ?y) (and (parent ?x ?y) (male ?x)))
(←  (= ?x ?y))
(←  (sibling ?x ?y) (and (parent ?z ?x)
                         (parent ?z ?y)
                         (not (= ?x ?y))))
```

то зможемо зробити наступні висновки:

```
> (with-answer (father ?x ?y)
(format t "~A is the father of ~A.~%" ?x ?y))
DONALD is the father of DEBBIE.
DONALD is the father of NANCY.
NIL
```

```
> (with-answer (sibling ?x ?y)
(format t "~A is the sibling of ~A.~%" ?x ?y))
DEBBIE is the sibling of NANCY.
NANCY is the sibling of DEBBIE.
NIL
```

Може здатися, що написаний нами код є простим і природнім рішенням поставленого завдання. Насправді, він вкрай неефективний. Насправді, ми написали, по суті, інтерпретатор, в той час як могли створити компілятор.

Наведемо начерк того, як це може бути зроблено. Основна ідея полягає в запаковування всієї програми в макроси **`<-`** і **`with-answer`**. У такому випадку основна частина роботи буде виконуватися на етапі компіляції, тоді як зараз вона виконується безпосередньо під час запуску. Будемо представляти правила як функції, а не як списки. Замість функцій типу **`prove`** і **`prove-and`**, інтерпретує вирази в процесі роботи, у нас будуть функції для перетворення виразів в код. Вирази стають доступні в момент визначення правила. Навіщо чекати, поки вираз буде використано, щоб його проаналізувати? Це відноситься і до **`with-answer`**, який буде викликати функції типу **`<-`** для генерації свого розкриття.

Здається, що подібна програма буде значно складніше наведеної в цій лекції, але на ділі реалізація запропонованої ідеї займе всього в два-три рази більше часу. Студентам, які бажають дізнатися більше про подібні методиках, рекомендується подивитися книгу *Peter Norvig «Paradigms of Artificial Intelligence Programming», Morgan Kaufman, 1992*. Ця книга, також відома як PAIP, розглядає програмування задач штучного інтелекту. Автор використовує Common Lisp і супроводжує книгу великою кількістю коду, який до ступен за адресою: http://norvig.com/paip/README.html .
Взяти книгу можна за посиланням - https://www.twirpx.com/file/674379/

Paradigms of AI Programming Source Code

This page is the index for the Lisp source code files for the book *Paradigms of Artificial Intelligence Programming*. Th

Installation Instructions

1. Download the file paip.zip and unzip it.
2. You must have a lisp compiler/interpreter.
3. To test all the code, start lisp and do the following at the interactive prompt:

```
(load "auxfns.lisp")
(requires "examples")
(do-examples :all)
```

   This should print out a long list of inputs and outputs, and the last output should be the total number of errors. I

Use

To use the code, edit any of the files or add new files. You will always have to do `(load "auxfns.lisp")` first, and you will t

The function "requires" is used for a primitive form of control over what files require other files to be loaded first. If "
complicated use of these files, you should follow the guidelines for organizing files explained in Chapter 24.

The function `do-examples`, which takes as an argument either `:all` or a chapter number or a list of chapter numbers, can

The Files

The index below gives the chapter in the book, file name, and short description for each file.

CH	Filename	Description
-	README.html	This file: explanation and index
-	examples.lisp	A list of example inputs taken from the book
-	tutor.lisp	An interpreter for running the examples
24	loop.lisp	Load this first if your Lisp doesn't support ANSI LOOP
-	auxfns.lisp	Auxiliary functions; load this before anything else
1	intro.lisp	A few simple definitions
2	simple.lisp	Random sentence generator (two versions)
3	overview.lisp	14 versions of LENGTH and other examples
4	gps1.lisp	Simple version of General Problem Solver
4	gps.lisp	Final version of General Problem Solver
5	eliza1.lisp	Basic version of Eliza program
5	eliza.lisp	Eliza with more rules; different reader
6	patmatch.lisp	Pattern Matching Utility
6	eliza-pm.lisp	Version of Eliza using utilities

Також можна порекомендувати книгу:

Остроух А.В. Введение в искусственный интеллект. Монография. — Красноярск: Научно-инновационный центр, 2020. — 250 с. — ISBN 978-5-907208-26-1.

https://www.twirpx.com/file/3567969/

У монографії викладені концептуальні засади і методи представлення знань в системах штучного інтелекту. Розглянуто різні підходи, що застосовуються при проектуванні і розробці інтелектуальних систем і технологій в транспортному комплексі, а також розглянуті тенденції розвитку систем штучного інтелекту.

ПРОЛОГ — мова програмування штучного інтелекту

Пролог (Prolog, програмування в логіці) - одна з найбільш широко використовуваних мов логічного програмування. Як і для інших декларативних мов, при роботі з нею ми описуємо ситуацію (правила й факти) і формулюємо мету (запит), дозволяючи інтерпретаторові Пролога знайти рішення задачі за нас.

Під інтерпретатором Прологу ми будемо розуміти механізм вирішення задачі за допомогою мови Пролог. Інакше кажучи, інтерпретатор мови Пролог - це виконавець Пролог-програм, тобто та "активна сила", що виконує програми, написані на Пролозі.

У кожної з мов програмування є своє коло задач, при вирішенні яких він використається з найбільшою ефективністю. Для Прологу це задачі, пов'язані з розробкою систем штучного інтелекту (різні експертні системи, програми - перекладачі, інтелектуальні ігри). Він використається для обробки природної мови й має потужні засоби, що дозволяють витягати інформацію з баз даних, причому методи пошуку, використовувані в ньому, принципово відрізняються від традиційних.

Пролог знайшов застосування й у ряді інших областей, наприклад, при вирішенні задач складання складних розкладів. При цьому він не є універсальною мовою програмування й не призначений, наприклад, для вирішення задач, пов'язаних із графікою або чисельними методами. Існує велика кількість реалізацій мови Пролог, як комерційних, так і вільно розповсюджених. Ми будемо орієнтуватися на SWI-Prolog та GNUProlog, розроблений в університеті міста Амстердам. Можливостей даної реалізації цілком достатньо для первісного знайомства з основами логічного програмування в межах програмування інтелектуальних інформаційних систем.

SWI-Prolog та GNU-Prolog поширюється під ліцензією GPL, що забезпечує можливість його використання без порушень або комерційних інтересів. Ця версія мови Пролог доступний як користувачам ОС Linux, так і користувачам Windows.

Логічні мови, як можна побачити з їхньої назви, для мети передачі змісту програм використають засоби математичної логіки. Сама по собі логіка була винайдена як інструмент людської думки, що дозволяє впорядкувати знання й одержати з них відповідні висновки. Тому ідея використання принципів математичної логіки при складанні комп'ютерних програм здається досить природньою.

Раніше в дисципліні «Дискретна математика» ми вже познайомилися із частиною логіки, яка зветься численням висловлювань. Але обрахування висловлювань не дає можливості виразити багато фактів та міркувань, якими користуються в повсякденному житті.

Наприклад, розглянемо класичне міркування:

Всі люди смертні (p);

Сократ - людина (q);

отже, (->)

Сократ смертний (r).

Це міркування вірне, але його неможливо довести в рамках теорії висловлювань. Ми можемо записати формулу **(p&&q)->r**, але довести її істинність уже не зможемо. Таким чином, логіка висловлень не дозволяє досить точно виразити розглянуте міркування. Це пов'язане з тим, що вона розглядає кожне висловлювання як неподільний об'єкт, у той час як багато висловлень залежать від якихось параметрів.

Числення предикатів є узагальненням числення висловлень, що дозволяють використати параметри (які також звуться аргументами або змінними) у висловленнях. У термінах теорії предикатів наше міркування можна записати так:

Для всіх x, якщо x є людиною,
те x є смертним;
Сократ є людиною;
(отже)
Сократ є смертним.

Вивчення числення предикатів не є нашою задачею, однак, для того, щоб застосовувати мову логічного програмування, не обов'язково знати логіку предикатів: вона вже вбудована в мову. Досить вивчити саму мову й звикнути до її виразних засобів.

Мова Пролог, найвідоміша із представників сімейства мов логічного програмування, вона зросла з робіт Алана Колмерауэра (A. Colmerauer) по обробці природної мови й незалежних робіт Роберта Ковальского (R. Kowalski) по використанню логіки у програмуванні. Дэвиду Уоррену (D. Warren) і його колегам з Эдинбургского університету вдалося здійснити досить ефективну реалізацію Прологу. Ім'я Уоррена ввійшло в історію логічного програмування.
У його честь названа базова техніка реалізації Прологу, що одержала назву **абстрактної машини Уоррена**.

Програма мовою Пролог являє собою набір фактів й (можливо) правил.

Якщо програма містить тільки факти, то її називають **базою даних**. Якщо вона містить ще й правила, то часто використають термін **база знань**.

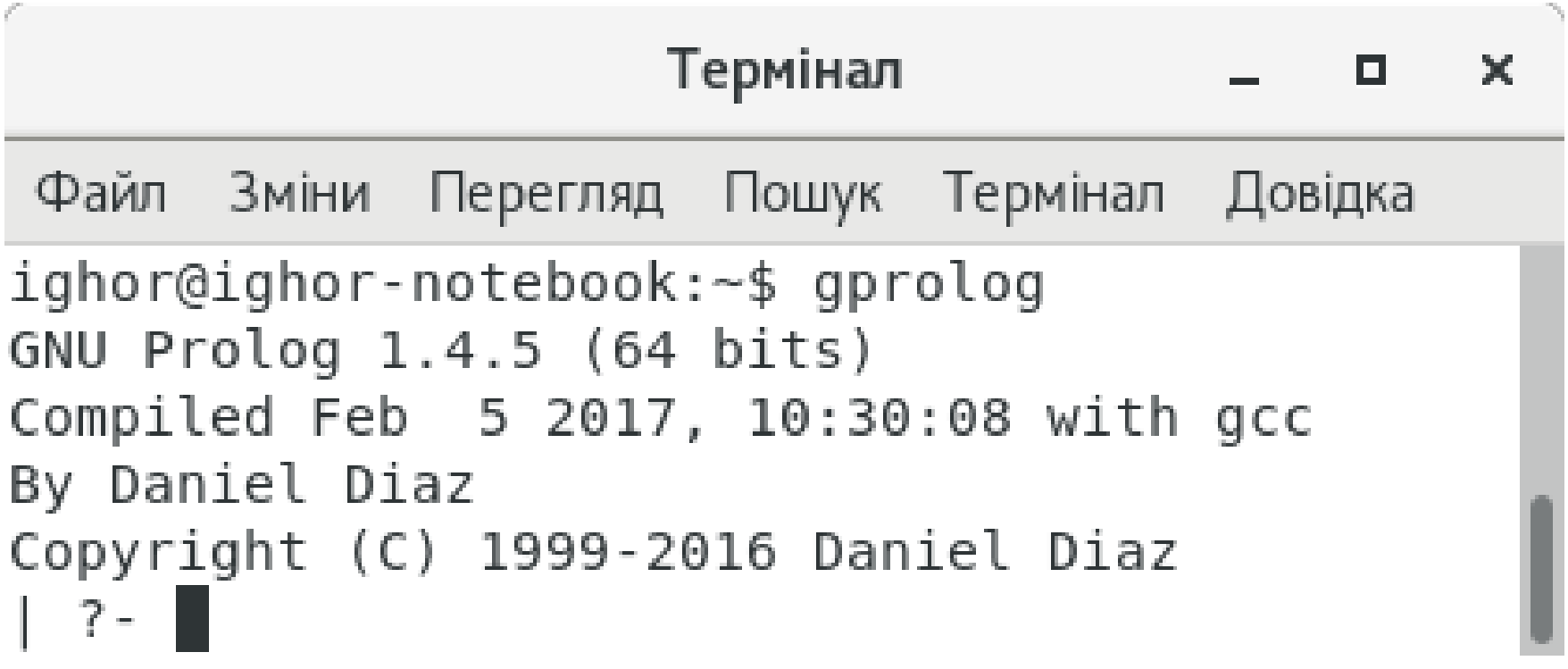
Інсталяція в терміналі системи Linux Ubuntu:

```
sudo apt update
sudo apt install gprolog
```

Після нормального відпрацювання інсталяції для запуску Прологу, наберіть у командному рядку `gprolog` і натисніть **Enter**.

На екрані з'явиться запрошення для введення запитів:

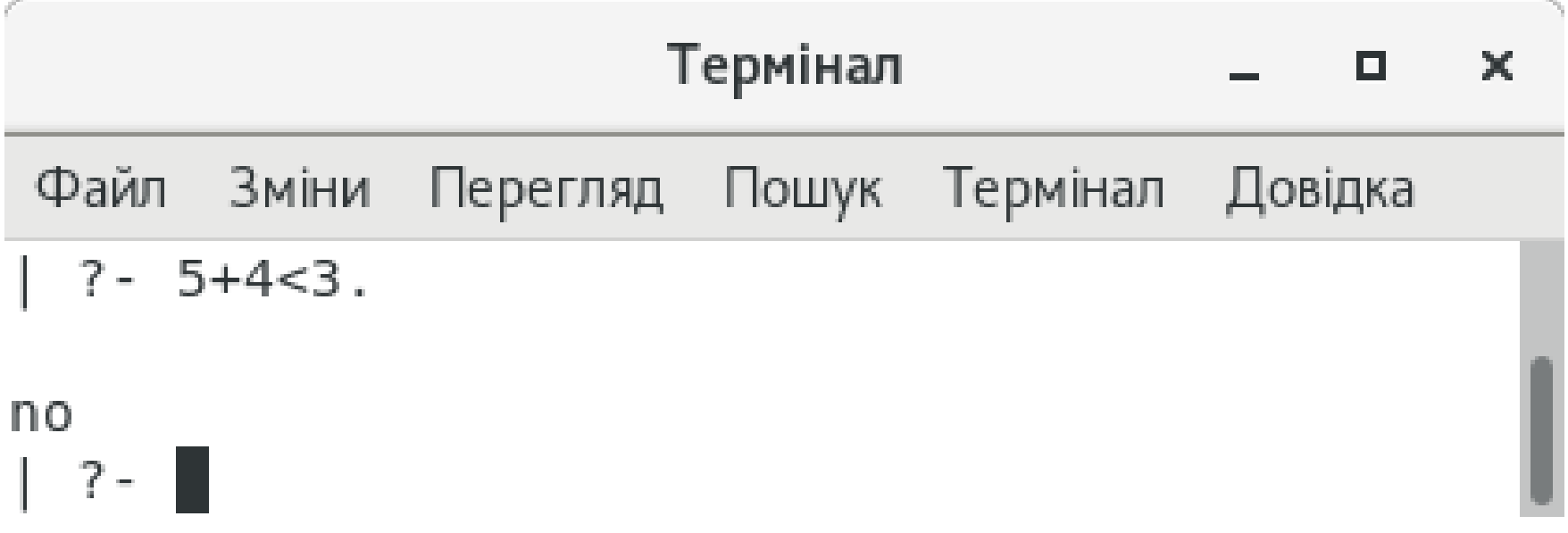
```
?-
```


Термінал
Файл Зміни Перегляд Пошук Термінал Довідка
ighor@ighor-notebook:~$ gprolog
GNU Prolog 1.4.5 (64 bits)
Compiled Feb 5 2017, 10:30:08 with gcc
By Daniel Diaz
Copyright (C) 1999-2016 Daniel Diaz
| ?-

Запит (питання) вводиться після запрошення й обов'язково закінчується крапкою, наприклад,

```
?- 5+4<3.
No
```

Пролог аналізує запит і видає відповідь **Yes** (Так) у випадку істинності твердження й **No** (Ні) у противному випадку або коли відповідь не може бути знайдена.



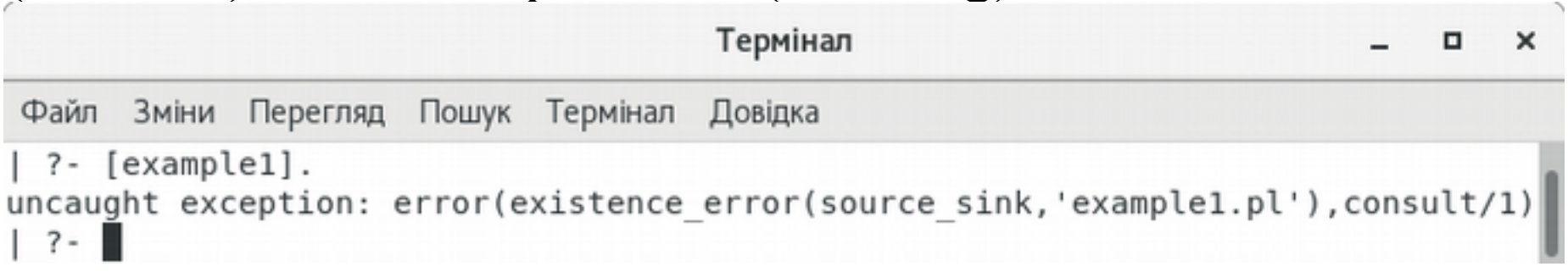
Зберігають програми мовою Пролог у текстових файлах, що найчастіше мають розширення **.pl**, наприклад, **example1.pl**. Для того щоб Пролог міг оперувати інформацією, що розміщена у файлі, він повинен ознайомиться з його вмістом (проконсультуватися з ним). Це можна зробити декількома способами. При використанні першого варіанта у квадратних дужках записується ім'я файлу (без **pl**), наприклад,

```
?- [example1].
```

У випадку вдалого завершення цієї операції буде видане повідомлення, аналогічне наступному:

```
% example1 compiled 0.00 sec, 612 bytes
Yes
```

У противному випадку буде виданий список помилок (ERROR) і/або попереджень (Warning).



Другий спосіб базується на виклику вбудованого предиката **consult**, якому як аргумент передається ім'я файлу (також без розширення), наприклад:

```
?- consult(example1).
```

Розширення **pl** часто використається для файлів, що містять програми мовою програмування Perl, тому можна зустріти й інші розширення для файлів із програмами на Пролозі. Для завантаження файлів з розширеннями, відмінними від **pl**, все ім'я файлу варто обов'язково вкласти в апострофи:

```
?- consult('example2.prolog').
?- ['example2.prolog'].
```

Обидві ці команди додають факти й правила із зазначеного файлу в базу даних Прологу. Можна завантажувати кілька файлів одночасно. У цьому випадку вони додаються через кому, наприклад,

```
?- [example1, 'example2.prolog'].
```

Важливо пам'ятати, що всі запити повинні закінчуватися крапкою. Якщо ви забудете її поставити, то Пролог виведе символ '|' і буде очікувати подальшого введення. У цьому випадку треба ввести крапку й нажати клавішу Enter:

```
?- [example1]
| .
Yes
```

Терми та об'єкти

Програма мовою Пролог зазвичай описує якусь дійсність. **Об'єкти** (елементи) описуваного світу представляються за допомогою термів. Терм інтуїтивно означає об'єкт. Існує 4 види термів: атоми, числа, змінні й складні терми. Атоми й числа іноді групують разом і називають найпростішими термами.

Атом - це окремий об'єкт, що вважається елементарним. У Пролозі атом представляється послідовністю літер нижнього й верхнього регістру, цифр та символів підкреслення '_', яка починається з малої літери. Крім того, будь-який набір припустимих символів, вкладений в апострофи, також є атомом. Нарешті, комбінації спеціальних символів `+ - * = < > : &` також є атомами (слід зазначити, що набір цих символів може відрізнятися в різних версіях Прологу).

Приклад

Представлені далі послідовності є коректними атомами:

```
b, abcXYZ, x_123, efg_hij, оля, слюсар,
'Це також атом Прологу',
+, ::, <---->, ***
```

Числа в Пролозі бувають цілими (Integer) і дробовими (Float). Синтаксис цілих чисел простий, як це видно з наступних прикладів: 1,1313, 0, -97. Не всі цілі числа можуть бути представлені в машині, їхній діапазон обмежений інтервалом між деякими мінімальним і максимальним значеннями, які визначаються певною реалізацією Прологу. SWI-Prolog допускає використання цілих чисел у діапазоні від -2147483648 (-231) до 2147483647 (231-1).

Синтаксис дробових чисел також залежить від конкретної реалізації. Ми будемо дотримуватися простих правил, зрозумілих з наступних прикладів: 3.14, -0.0035, 100.2. При звичайному програмуванні на Пролозі дробові числа використаються рідко. Причина цього в тім, що Пролог - мова, призначена в першу чергу для обробки символьної, а не числової інформації. При символьній обробці часто використовуються цілі числа, потреба в дробових числах невелика. Скрізь, де можна, Пролог намагається привести число до цілого виду.

Змінними в Пролозі є рядки символів, цифр і символу підкреслення, що починаються із великої букви або символу підкреслення:

```
X, _4711, X_1_2, Результат, _x23, Об'єкт2, _
```

Останній приклад (єдиний символ підкреслення) є особливим випадком - анонімної змінної (змінної без імені). Анонімна змінна застосовується, коли її значення не використається в програмі. Можливо кількаразове вживання безіменної змінної в одному виразі застосовується для того, щоб підкреслити наявність змінних при відсутності їхньої специфічної значимості.

Складні терми (функції) складаються з імені функції (нечислового атома) і списку аргументів (термів Прологу, тобто атомів, чисел, змінних або інших складових термів), що взяті у круглі дужки й розділені комами. Групи складних термів використовують для складання фраз Прологу. Не можна розміщувати символ пробілу між функтором (ім'ям функції) і відкриваючою круглою дужкою. В інших позиціях, однак, пробіли можуть бути корисні для легшого читання програм. Нижче наведено два складні терми:

```
итого(клієнт(X,23,_), 71)
'Що трапилося?'(нічого)
```

При завданні імен термів переважніше використати мнемонічні імена, тому що терм `а(ж)`, наприклад, набагато менш інформативний, ніж терм `автор(жуль_верн)`.

Ще однією важливою структурою даних у Пролозі є список. Ми познайомимося з ним пізніше. Зараз відзначимо тільки один з видів списків - список символів.
Такі списки можуть бути представлені у вигляді **рядків**, наприклад, перший аргумент складного терму `вік("Борис",10)` - рядок. При записі рядки беруться у лапки.

Програмувати на Пролозі - означає описувати якийсь світ. Програма на цій мові складається із множини **фраз**, що задають взаємозв'язок між термами. Кожен терм позначає ту або іншу сутність, що належить світу. Один зі способів опису - це завдання **фактів**.

Факт - це твердження про те, що існує деяке конкретне відношення. Він є безумовно вірним. У розмовній мові під фактом розуміється вислів типу "Сьогодні сонячно" або "Викладачу 64,5 роки". На Пролозі це записується у вигляді

```
'Сьогодні сонячно'.
'Викладачу 64,5 роки'.
```

Якщо ви збережете ці факти у файлі й потім завантажите його, то можна задавати питання інтерпретаторові Прологу (нагадаємо, що запит вводиться після запрошення Прологу, яке у більшості версій має вигляд **?-**) , наприклад,

```
?- 'Сьогодні сонячно'.
Yes
?- 'Викладачеві 64,5 роки'.
Yes
?- 'Сьогодні сонячно', 'Викладачеві 64,5 роки'.
Yes
```

Кома між фактами в останньому запиті означає операцію логічного **&** (кон'юнкцію).

Така форма запису відповідає логіці висловлень, можливості якої, як уже говорилося, досить обмежені. Ми не можемо задати, наприклад, питання про те, скільки років Васі. Набагато зручніше використати параметризовані факти, роботу з якими підтримує логіка предикатів. На Пролозі факт може бути записаний у вигляді предиката, аргументи якого є символьними або числовими константами.

У загальному випадку предикат - це логічна функція від одного або декількох аргументів, тобто функція, що діє в множині з двох значень: істина та неправда. Предикат Прологу записується у вигляді складного терму:

```
ім'я_предикату(аргументи).
```

Аргументи перераховуються через кому та являють собою якісь об'єкти або властивості об'єктів, а ім'я предиката позначає зв'язок або відношення між аргументами. Предикат однозначно визначається парою: ім'я та кількість аргументів. Два предикати з однаковим ім'ям, але різною кількістю аргументів, вважаються різними. Кількість параметрів предиката називається його арністю (arity). При описі предиката арність вказують після його імені, розділяючи їх символом '/' (слеш). Як правило, імена предикатів й аргументів записуються в називному відмінку. Пробіли в них не допускаються, тому як роздільник в символьних константах використається символ підкреслення.

Приклад

Факт "Микола працює слюсарем" на Пролозі запишеться у такий спосіб:

```
професія(микола, слюсар).
```

Тут предикат професія має два аргументи: перший означає ім'я людини, а

другий - професію. Факт "Борису 10 років" можна представити у вигляді:

```
вік("Борис", 10).
```

Порядок аргументів предиката зв'язаний зі змістом факту й тому не змінюємо.

При записі фактів треба пам'ятати, що:

```
ім'я факту починається з малої літери;
запис кожного факту закінчується крапкою.
```

У наведених вище прикладах професія та вік - предикати (складні терми), микола та слюсар - атоми, 10 - число, "Борис" - рядок. Докладніше про види термів Прологу розповідається далі.

База даних на Пролозі - це сукупність фактів. У процесі роботи в базу даних можна додавати нові факти, видаляти або змінювати старі.

Приклад

Складемо базу даних з наступних фактів:
"слон більше, ніж кінь",
"кінь більше, ніж віслюк",
"віслюк більше, ніж собака",
"віслюк більше, ніж мавпа":

```
більше(слон, кінь).
більше(кінь, віслюк).
більше(віслюк, собака).
більше(віслюк, мавпа).
```

Ми використали предикат **більше**, що має два параметри. Збережемо цю базу даних у текстовому файлі та потім познайомимо Пролог з нею. Тепер можна формулювати запити до інтерпретатора Прологу:

```
?- більше(слон, кінь).
Yes
?- більше(кінь, слон).
No
```

Запит - це послідовність предикатів, які розділені комами та завершені крапкою. Природною мовою кома відповідає союзу "і", а мовою математичної логіки позначає кон'юнкцію. За допомогою запитів можна "запитувати" базу даних про те, які твердження є істиними. Предикат запиту зветься **метою**.

Прості питання, що не містять ніяких змінних, називають **та-ні- питаннями**.

Вони допускають лише дві можливі відповіді: "**Yes**" означає наявність відповідного факту в базі даних (перший запит приклада, наведеного нижче), "**No**" - його відсутність (другий запит). У випадку відповіді "**Yes**" говорять, що запит завершився успіхом, ціль досягнута.

Приклад

```
?- більше(слон, кінь), більше(кінь,вислюк).
Yes
?- більше(слон, собака).
No
```

Використання змінних у запитах дозволяє задавати більш складні питання. Припустимо, наприклад, що ми хочемо визначити, які тварини більше вислюка. У наступному запиті змінна `X` позначає шукану відповідь:

```
?- більше(X, вислюк).
X = кінь
Yes
```

При обробці запиту змінна `X` прийняла значення "кінь". Переглядаючи базу даних, інтерпретатор виявив факт, що стверджує, що кінь більше віслюка, та запит був успішно виконаний.

Запити зі змінними можуть мати більше одного рішення. Першим завжди виводиться те з рішень, що знаходиться ближче до початку бази даних. Якщо нам досить тільки однієї відповіді, то можна нажати Enter і закінчити пошук. У випадку, якщо ми захочемо одержати чергову відповідь, потрібно нажати клавішу ; (крапка з комою), і Пролог почне пошук інших варіантів відповіді на запит. Повідомлення "`No`" говорить про відсутність чергового рішення.

Приклад

```
?- більше(вислюк, X).
X = собака;
X = мавпа;
No
?- більше(X,Y).
X = слон
Y = кінь;
X = кінь
Y = вислюк;
X = вислюк
Y = собака;
X = вислюк
Y = мавпа;
No
```

Наступна лекція буде присвячена створенню баз знань на мові Prolog.