Програмування інтелектуальних інформаційних систем

3 курс, осінь 2021

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

Лекція 13

Приклади програмування розв’язання задач в системі CLIPS

Приклад №1 програмування задачі на CLIPS

Розглянемо предметну область, яка представляє учасників певної конференції, які приїхали з різних міст. На таких заходах усі учасники зазвичай проходять реєстрацію. Нехай ця процедура є введенням відомостей про учасників до бази даних, в якій на кожного учасника виділяється один запис (факт), що складається зі списку з трьома полями. Нехай перше поле має символьне значення **`rep`** – скорочення від **`representative`** (представник).

Загалом це значення може бути будь-яким, а поле може бути відсутнім. У другому полі списку зберігається прізвище учасника, а третьому – місто, з якого учасник прибув.

Вміст фактів бази даних з ім'ям rep може бути, наприклад, таким:

```
(deffacts rep
(rep Alejnov odesa)
(rep Ladak odesa)
(rep Slobodjanjuk lviv)
(rep Klitka lviv)
(rep Bojko kyiv)
(rep Pustovit odesa)
(rep Spokojnij odesa)
(rep Shamis odesa)
(rep Lobovko kyiv)
(rep Zadorozhna lviv)
(rep Javorskij lviv))
```

Використовуючи будь-який текстовий редактор, створимо та збережемо базу даних у вигляді текстового ASCII-файлу з ім'ям, що повторює ім'я бази даних (тобто rep). Це дозволяє легко редагувати дані, незалежно від інших програмних модулів, додаючи нових учасників або видаляючи вибулих.

Після закінчення конференції організатори підбивають підсумки, визначаючи масу показників. Зокрема, нехай потрібно визначити кількість представників кожного міста. Алгоритм розв'язання такого завдання простий. Для кожного міста задаємо лічильник та послідовно переглядаємо списки в записах файлу **`rep`**. Якщо в записі третє поле списку має значення **`kyiv`**, вміст відповідного лічильника збільшуємо на одиницю. Для інших міст – аналогічно.

Програма мовою CLIPS, що реалізує зазначений алгоритм, може бути, наприклад, такою:

```
(defglobal ?*odesa* = 0)
(defglobal ?*kyiv* = 0)
(defglobal ?*lviv* = 0)
(defrule start
(initial-fact)
=>
(printout t crlf «REPRESENTATIVES» crlf)
(defrule odesa
(rep ? odesa)
=>
(bind ?*odesa* (+ ?*odesa* 1)))
(defrule kyiv
(rep ? kyiv)
=>
(bind ?*kyiv* (+ ?*kyiv* 1)))
```

```
(defrule lviv
(rep ? lviv)
=>
(bind ?*lviv* (+ ?*lviv* 1)))
(defrule result
(declare (salience -1))
(initial-fact)
=>
(printout t «from odesa: « ?*odesa* crlf)
(printout t «from kyiv: « ?*kyiv* crlf)
(printout t «from lviv: « ?*lviv* crlf))
```

У перших трьох рядках програми за допомогою конструктора **defglobal** оголошуються три глобальні змінні: **?*odesa***, **?*kyiv*** та **?*lviv***. Ці змінні є лічильниками. У CLIPS змінна може бути локальною – але тоді вона пов'язується лише з тим правилом, у якому оголошується.

Далі йде правило з ім'ям **start**, ліва частина якого є запис **(initial-fact)**. Так позначається початковий фактний факт, який створюється в робочій пам'яті інтерпретатора CLIPS за командою **(reset)** до запуску програми на виконання. Але ж навіщо він потрібний?

Справа в тому, що в CLIPS-програмах поширеними правилами є такі, які додають факти до бази даних, або, навпаки, їх видаляють. Типовою є ситуація, коли при старті програми у базі даних немає фактів, що задовольняють хоча б одному правилу. У цьому випадку програма нічого не виконає. Для того щоб почати обчислення та використовується системний початковий факт, який, незалежно від фактів у базі даних, активізує деяке правило, що додає такі факти, які, своєю чергою, активізують правила, умови яких не виконувались у початковий момент.

У цій програмі **(initial-fact)** запускає правило **start**, яке активізується незалежно від фактів у файлі **rep** і є у програмі лише з єдиною метою – вивести заголовок. Для цього в його правій частині викликається вбудована функція **printout** з ключем **t**, що виводить на стандартний пристрій виведення (монітор) заголовок, укладений у лапки. Комбінація символів **crlf** є аналогом **endl** C++ і служить для переведення курсору на наступний рядок.
Наступні три правила з іменами **odesa**, **kyiv** та **lviv** можна назвати ядром програми. У них проводиться підрахунок кількості учасників – відповідно з Одеси, Києва та Львова.

Розглянемо правило **lviv**. Воно активізується у разі, як у базі даних перебуває факт **(rep ? lviv)**. Неважко здогадатися, що символ **"?"** У другому полі цього списку означає символ універсальної підстановки і замінює собою будь-яке прізвище. Звідси випливає, що правило **lviv** активізується стільки разів, скільки разів факт **(rep? lviv)** є у базі даних. При цьому стільки ж разів виконаються дії, що містяться у правій частині правила. Вбудована функція **bind** – аналог оператора присвоєння. Отже, вміст змінної **?*lviv*** збільшується на одиницю і результат зберігається у цій змінній.
Аналогічно працюють правила **odesa** та **kyiv**.

Дії, які виконуються в останньому правилі програми, відображені у його назві. Права частина правила особливих коментарів не вимагає, у той час як ліва частина заслуговує на докладний розгляд. У CLIPS існує кілька стратегій черговості виконання правил, а самі правила можуть мати пріоритет, який задається вбудованою функцією **`declare`** із параметром **`salience`** (особливість). Цей параметр може набувати цілих чисел від **`-10000`** до **`+10000`**. За промовчанням всім правил величина **`salience`** дорівнює нулю.

Якщо у правилі **`result`** не вказати пріоритет, воно буде конфліктувати з правилом **`start`** за черговість виконання, оскільки ці правила мають однакову ліву частину. Для усунення конфлікту у правилі **`result`** пріоритет зазначений явно і зі знаком мінус, у зв'язку з чим це правило виконається останнім.

Використовуючи будь-який текстовий редактор, наберемо та збережемо текст програми в ASCII-файлі зі стандартним для CLIPS-програм розширенням **`.clp`** та з ім'ям **`represent`**. Командою **`clips`** викличемо інтерпретатор CLIPS, командою **`(load имя_файла)`** завантажимо в інтерпретатор файли **`rep`** і **`represent.clp`**, командами **`(reset)`** та **`(run)`** запустимо програму **`represent.clp`** на виконання.

```
CLIPS (V6.21 06/15/03)
CLIPS> (load rep)
.........
TRUE
CLIPS> (load represent.clp)
.........
TRUE
CLIPS> (reset)
CLIPS> (run)
REPRESENTATIVES
from odesa: 5
from kyiv: 2
from lviv: 4
CLIPS>
```

Повідомлення інтерпретатора **TRUE** означає, що файл не має синтаксичних помилок і команда завантаження виконана коректно.
Многоточием представлені інші повідомлення інтерпретатора, які у цьому разі опущені.

Як випливає з описаних дій, в інтерпретаторі CLIPS є два файли. Перший з ім'ям **`rep`** є базою даних. Другий, з іменем **`represent.clp`**, містить відомості (правила) про те, як ці дані можуть бути використані. Таким чином, разом файли утворюють базу знань, яка містить принаймні два знання. Перше – загальний склад учасників конференції. Його можна подивитися, не виходячи з інтерпретатора за командою **`(facts)`**. Друге знання – кількість учасників кожного міста.

У розглянутому прикладі база знань і двох програмних модулів. Однак ніщо не заважає використовувати одну програму, збережену в одному файлі. У наведеному нижче прикладі показано, як це робиться. У ньому ж евристичний механізм уявлення знань використовується разом із процедурним.

Приклад №2 програмування задачі на CLIPS

Нехай потрібно підібрати резистор для ділянки ланцюга схеми електричної принципової деякого радіоелектронного пристрою. Резистор характеризується опором, який визначається за виміряними або розрахованими значеннями електричного струму, що проходить через резистор і падіння напруги на ньому.

Програма з ім'ям **resistor.clp**, яка вирішує це завдання, може бути, наприклад, такою

```
(deffacts resistors; база даних резисторів
(Resistor Ra 2)
(Resistor Rb 5)
(Resistor Rc 7))
(deffunction om; функція om(x,y)
(? X? Y)
(div? y?x))
(defrule input; початкове правило
(initial-fact)
=>
(printout t crlf "Input current value: ")
(bind? i (read))
```

```
(printout t "Input strait value: ")
(bind ?u (read))
(assert (numbers ?i ?u)))
(defrule take; підібрати резистор із БД
(Numbers ?i ?u)
(resistor ?r = (om ?i ?u))
=>
(printout t crlf "Ви маєте дати resistor
"?r"." crlf crlf)
(Reset)
(Halt))
(defrule nothing; якщо в БД немає
відповідного резистора
(Numbers ?i ?u)
```

```
(resistor ?r ~=(om ?i ?u))
=>
(printout t crlf “There is nothing for You
in my database!” crlf
crlf)
(Reset)
(Halt))
```

Програма складається з декількох частин: бази даних з ім'ям **`Resistors`**, оголошення користувача функції om і трьох правил з іменами **`input`**, **`take`** і **`nothing`**.

У базі даних містяться відомості про резистори. Вони представлені у вигляді списків, що складаються із трьох полів.

Перше поле має значення **`resistor`**, яке відбиває тип радіодеталі. У другому полі списку міститься тип резистора. Останнє поле зберігає значення опору. Про функцію om детально говорилося раніше. В даному випадку вона використовується уявлення процедурного знання – закону Ома. Правило **`input`** призначене для введення вихідних даних. Воно активізується початковим системним фактом і вимагає від користувача ввести струм і напругу. Вбудована функція **`read`** повертає значення, введене зі стандартного пристрою введення (клавіатури), яке зберігається в змінних **`?i`** та **`?u`**.

У правій частині правила виконується ще одна дія. Команда **`assert`** додає в робочу пам'ять інтерпретатора CLIPS факт **`(numbers ?i ?u)`** для того, щоб можна було звертатися до локальних змінних **`?i`** та **`?u`**, пов'язаних із правилом **`input`**, з інших правил програми.

У наступних двох правилах користувачеві пропонується тип відповідного резистора (правило **`take`**), або повідомляється про відсутність такого (правило **`nothing`**).

Розглянемо правило таке. Його ліва частина складається з двох умов, тому правило активізується, якщо обидві умови будуть виконані. Перше умова виконується, оскільки відповідний факт вже створено правилом input. Друга умова виконається, якщо точно відповідатиме якомусь факту (списку) у базі даних.

Перше поле умови питань не викликає. У другому полі умови перемінна **?r**, яка може прийняти значення **Ra**, або **Rb**, або **Rc** - залежно від вмісту третього поля умови. У цьому полі здійснюється виклик функції om і зберігається значення, що повертається функцією. Так, якщо значення, що повертається дорівнюватиме **7**, то умова виконається, змінна **?r** прийме значення **Rc**, правило активізується і виведе на екран монітора пропозицію вибрати резистор **Rc**. Якщо значення om, що повертається функцією, дорівнює **5**, то користувачеві буде запропонований резистор **Rb** і т.д.

У лівій частині правила `nothing` як би повна аналогія – за винятком однієї невеликої модифікації. У третьому полі другого умови перед викликом функції om стоїть символ `" ~ "`, що означає логічне заперечення. Таким чином, умова виконається і правило активізується, якщо значення, що повертається функцією om значення буде не `2`, не `5` і не `7`.

Перебуваючи в інтерпретаторі CLIPS, командою **(clear)** очистимо його від даних попереднього прикладу, завантажимо файл **resistor.clp** і запустимо програму виконання:

```
CLIPS> (clear)
CLIPS> (load resistor.clp)
.............
TRUE
CLIPS> (reset)
CLIPS> (run)
Input current value: 3
Input strait value: 15
Ви повинні приймати Rb Rb.
CLIPS> (run)
```

```
Input current value: 3.5
Input strait value: 5.44
There is nothing for You in my database!
CLIPS>
```

Таким чином, файл **resistor.clp** також є базою знань, оскільки містить і базу даних, і відомості (правила) про те, як дані можуть бути використані. Ця база має, принаймні, три знання. Перше – загальний список резисторів із зазначенням типу та опору. Друге – закон Ома. Третє знання – пропонований тип резистора.

Інтелект бази знань можна суттєво підвищити додаванням нових даних та правил. Так, замість закону Ома можна використовувати серйозніші методики визначення опору резистора, наприклад схемотехнічну САПР PSpice. Результати її роботи можна зберегти в текстовому файлі, а потім викликати з третього поля другого правила правила таке.

Інший шлях – додавання нових типів резисторів до бази даних. Наприклад, цікавий результат виходить при внесенні до бази запису **`(resistor Rd 2)`**. При деякому доопрацюванні правил **`take`** і **`nothing`**, якщо значення **`om`**, що повертається функцією, дорівнює **`2`**, правило **`take`** відпрацює два рази і запропонує резистори **`Ra`** і **`Rd`**. Потім можна піти далі: додати правило (правила), яке вибере з резисторів **`Ra`** та **`Rd`** кращий для деяких конкретних умов тощо.

Задача “Правдолюбці та брехуни” на CLIPS

Для того, щоб продемонструвати можливості мови CLIPS візьмемо логічну головоломку.

У головоломці вирішується одне із завдань, що виникають на острові, населеному мешканцями двох категорій: одні завжди говорять правду (назвемо їх правдолюбцями), а інші завжди брешуть (їх, природно, назвемо брехунами). Безліч подібних головоломок ви можете зустріти на сторінках цікавої книги Раймонда Смуляна "What is the Name of this Book?". Нижче наведено різні завдання із цієї серії.

Р1. Зустрічаються дві людини, А і В, один з яких правдолюб, а інший - брехун. А каже: «Або я брехун, або У правдолюбець». Хто з цих двох правдолюбець, а хто брехун?

Р2. Зустрічаються троє людей, А, В і С. А й каже: «Усі ми брехуни», а В відповідає: «Тільки один із нас правдолюбець». Хто з цих трьох правдолюбець, а хто брехун?

Р3. Зустрічаються троє людей, А, В і С. Четвертий проходячи повз, запитує А: «Скільки правдолюбців серед вас?» А відповідає невизначено, а У відповідає: «А сказав, що серед нас є один правдолюбець». Тут у розмову вступає С і додає: «В бреше!» ким на вашу думку є В і С?

У програмі, яка вирішує проблеми такого класу, будуть використані широкі можливості засобів програмування правил у мові CLIPS та продемонстровані деякі цікаві прийоми, наприклад використання контекстів та зворотного відстеження. Ми також покажемо, як конструювати та тестувати прототипи, які приблизно відтворюють поведінку остаточної програми. Як зазначалося в основному матеріалі книги, технологія побудови експертних систем з використанням прототипів – одна з найпоширеніших на даний час.

Аналіз проблеми

Першим етапом будь-якого програмного проекту є аналіз розв'язуваної проблеми. Експерт має вміти вирішити проблему, а інженер із знань має розібратися, як саме було отримано рішення. При вирішенні нашого завдання вам доведеться виступити в обох іпостасях.

Запропоновані головоломки можна вирішити, систематично аналізуючи, що станеться, якщо персонаж, який вимовляє репліку, є правдолюбцем, а що якщо він – брехун. Позначимо через **`T(A)`** факт, що **`A`** говорить правду, а отже, є правдолюбцем, а через **`F(A)`** – факт, що **`A`** бреше і, отже, є брехуном.

Розглянемо спочатку головоломку **P1**. Припустимо, що говорить правду. Тоді з його репліки випливає, що або **A** брехун, або У правдолюб. Формально це можна уявити в наступному вигляді:

T(A) =>F(A) v T(B)

Оскільки **A** не може одночасно бути і брехуном, і правдолюбцем, то звідси випливає

T(A) =>T(B)

Аналогічно можна записати інший варіант. Припустимо, що **A** бреше:

F(A)=>-(F(A) v T(B)).

Спростимо цей вислів:

F(A)=>-F(A) ^-T(B) або F(A)=> T(A)^F(B).

Порівнюючи обидва варіанти, неважко дійти висновку, що тільки останній правильний, оскільки в першому варіанті ми дійшли висновку, що суперечить умовам (не можуть бути правдолюбцями одночасно **A** і **B**).

Таким чином, розглянута проблема відноситься до типу таких, вирішення яких знаходиться в результаті аналізу висновків, що випливають із певних припущень, та пошуку в них протиріч (або відсутності таких). Ми припускаємо, що певний персонаж говорить правду, а потім дивимося, чи можна в цьому випадку так розподілити ролі інших персонажів, що не будуть порушені умови, сформульовані в репліках.

На жаргоні, прийнятому в математичній логіці, припущення про правдивість чи брехливість безлічі висловлювань називається інтерпретацією, а варіант несуперечливого присвоєння значень істинності елементам множини – моделлю.

Однак наші головоломки включають і щось, що виходить за рамки типових проблем математичної логіки, оскільки репліки в них може вимовляти не один персонаж (як у головоломці **`P2`**), а на репліку одного персонажа може бути репліка у відповідь іншого (як в головоломці **`P3`**). У вихідній версії програми, яку ми розглянемо нижче, це ускладнення відсутнє, але остаточної воно має бути враховано.

Ми покажемо, що поступове ускладнення програми досить добре узгоджується з використанням правил. Насправді виявляється, що у першої версії програми найзручніше скористатися «виродженим» варіантом проблеми, тобто. постаратися вирішити її у тривіальному вигляді, який, тим не менш, несе у собі багато особливостей реального випадку. Ось як це виглядає щодо наших правдолюбців та брехунів.

P0. **A** заявляє: "Я брехун". Хто ж насправді **A** – брехун чи правдолюб?

Ми тільки-но фактично процитували добре відомий Парадокс Лгуна. Якщо брехун, то, отже, він бреше, тобто. насправді він правдолюб. Але тоді ми приходимо до суперечності. Якщо ж правдолюбець, тобто. каже правду, то насправді він брехун, а це знову суперечність. Отже, у цій головоломці немає несуперечливого варіанта «розподілу ролей», тобто. не існує моделі в тому сенсі, що надається їй у математичній логіці.

Є багато переваг у виборі для прототипу програми варіанта головоломки **`РО`**.

- У головоломці є лише один персонаж.
- Вираз не містить логічних зв'язок, таких як **`І`** або **`АБО`**, або кванторів, на кшталт квантора спільності (всі) та інших.
- Відсутня репліка у відповідь.

У той самий час істотні риси проблеми цьому варіанті присутні. Ми як і маємо спробувати знайти несуперечливу інтерпретацію висловлювання **A**, тобто. повинні реалізувати два завдання, присутні у будь-яких варіантах подібної головоломки:

- формувати альтернативні інтерпретації висловлюванням;
- аналізувати наявність протиріч.

Онтологічний аналіз та подання знань

Наступний етап – визначити, з якими видами даних нам доведеться мати справу під час вирішення цього класу головоломок. Які об'єкти становлять інтерес у світі правдолюбців та брехунів і якими атрибутами ці об'єкти характеризуються?

Очевидно, для вирішення завдань цього класу нам доведеться мати справу з такими об'єктами.

- Персонажі, які вимовляють репліки. Репліка, що вимовляється, характеризує або самого персонажа, або інших персонажів, або і тих, і інших. Персонаж може бути або правдолюбцем, або брехуном.

• Затвердження, що міститься у репліці. Це твердження може бути цілком брехливим, або абсолютно правдивим (істинним).

Трохи подумавши, ми дійдемо висновку, що існують ще й інші об'єкти, які необхідно враховувати під час вирішення завдань цього класу.

• існує середовище (світ), яке характеризується сукупністю наших припущень. Наприклад, існує світ, у якому ми припустили, що А – правдолюбець, а отже, висловлене ним твердження (або твердження) істинно. Це припущення спричиняє різні наслідки, які утворюють контекст даного гіпотетичного світу.

• Існує ще щось, що ми назвемо причинами або причинними зв'язками (reasons), які пов'язують висловлювання про той чи інший гіпотетичний світ. Якщо А стверджує, що «В – брехун», і ми припускаємо, що А – правдолюбець, то це твердження є причиною (підставою), через яку ми можемо стверджувати, що в даному гіпотетичному світі В – брехун, а отже, всі твердження, які містяться в репліках, що вимовляються, брехливі. Відслідковуючи такі зв'язки між висловлюваннями, можна відновити вихідний стан проблеми, якщо в результаті міркувань ми дійдемо протиріччя.

Звичайно, що ці об'єкти можна представляти в програмі по-різному. Онтологічний аналіз практично ніколи не призводить до єдиного способу уявлення.
Для першої версії CLIPS-програми було вибрано таке представлення описаних об'єктів:

```
;;;    Об'єкт    statement    (висловлювання)
пов'язаний із певним
;;; персонажем (поле speaker).
;;; Вислів містить твердження (поле claim).
;;; Висловлювання має підставу – причину (поле
reason),
;;; за якою йому можна довіряти,
;;; та тег (tag) – це може бути довільний
;;; ідентифікатор.
(deftemplate statement
     (field speaker (type SYMBOL))
      (multifield claim (type SYMBOL))
   (multifield reason (type INTEGER) (default
0))
        (field tag (type INTEGER) (default 1))
)
```

Замість того, щоб фокусувати увагу на персонажі, на чільне місце ставимо вимовлену ним репліку (висловлювання), а персонаж відносимо до атрибутів висловлювання. Хочемо забезпечити можливість подати певну головоломку у вигляді екземпляра шаблону, наведеного нижче.

```
(statement (speaker A) (claim F A))
```

Цей шаблон можна перекласти на «людську» мову таким чином:
«Існує висловлювання, зроблене персонажем **A**, у якому стверджується, що **A** брехун і тег цього висловлювання за умовчанням набуває значення **1**». Зверніть увагу, що в полі reason також буде встановлено значення за промовчанням (це значення дорівнює **0**), тобто. ми можемо припустити, що жодних попередніх висловлювань, які б підтвердити дане, цього завдання було.
Зверніть увагу, що поля **claim** та **reason** мають кваліфікатор **multifield**, оскільки вони можуть містити декілька елементів даних.

Однак недостатньо тільки уявити в програмі висловлювання персонажів – нам знадобиться також виявити суть тверджень, що містяться в них. Далі, прийнявши певне припущення про правдивість чи брехливість персонажа, якому належить висловлювання, можна побудувати гіпотезу про істинність чи брехливість цього твердження. З кожним таким твердженням зв'яжемо унікальний числовий ідентифікатор.

```
;;; Твердження, сенс якого, наприклад,
;;; полягає в наступному,
;;; T A . . . . означає, що A правдолюбець;
;;; F A . . . . означає, що A брехун.
;;; Твердження може мати під собою
;;; основа (reason) – зазвичай це тег
;;; висловлювання (об'єкта statement) або тэг
;;; іншого затвердження (об'єкта claim).
;;; Твердження також характеризується ознакою
scope,
;;; який може набувати значення «істина» або
«брехня».
```

```
(deftemplate claim
    (multifield content (type SYMBOL))
     (multifield reason (type INTEGER)
(default 0))
            (field scope  (type SYMBOL))
)
```

Наприклад, розкривши вміст наведеного вище висловлювання у припущенні, що **A** говорить правду, отримаємо такі твердження **(об'єкт claim)**:

```
(claim (content F A) (reason 1) (scope truth)).
```

Таким чином, об'єкт **`claim`** успадковує вміст від об'єкта **`statement`**. Останній стає обґрунтуванням **`(reason)`** цього твердження. Поле **`scope`** об'єкта **`claim`** приймає значення припущення про правдивість чи брехливість цього висловлювання.

Ще нам знадобиться представлення в програмі того світу (**`world`**), в якому ми зараз перебуваємо. Об'єкти **`world`** породжуються у момент, коли ми формуємо певні припущення. Потрібно мати можливість розрізняти різні множини припущень та ідентифікувати їх у програмі в той момент, коли процес роздумів приводить нас до суперечності. Наприклад, протиріччя між висловлюваннями **`T(A)`** і **`F(A)`** відсутня, якщо вони істинні у різних світах, тобто. за різних припущень.

Світи представлятимемо у програмі наступним чином:

```
;;; Об'єкт world представляє контекст,
;;; сформований певними припущеннями
;;; про правдивість чи брехливість персонажів.
;;; Об'єкт має унікальний ідентифікатор у полі
tag,
;;;  а  сенс  припущення  –  істинність  чи
брехливість –
;;; фіксується у полі scope.
(deftemplate world
              (field tag (type INTEGER)
(default 1))
              (field scope (type SYMBOL)
(default truth))
)
```

Зверніть увагу на те, що при вказаних у шаблоні значеннях за умовчанням ми можемо починати кожен процес обчислень з об'єкта world, що має в полі значення 1, причому цей світ можна заселити висловлюваннями персонажів, яких ми імовірно вважаємо правдолюбцями. Таким чином, можна ініціалізувати базу фактів the-facts для задачі P0 наступним чином:

```
;;; Твердження, що A брехун.
(deffacts the-facts
          (world)
                    (statement (speaker A)
(claim F A))
)
```

Якщо цей оператор **`deffacts`** буде включений у той же файл, що і оголошення шаблонів (а також правила, про які йтиметься нижче), то після завантаження цього файлу в середу CLIPS нам знадобиться для запуску програми дати лише команду **`reset`**.

Разработка правил

Тепер ми розглянемо набір правил, який допомагає впоратися з виродженим формулюванням **`P0`** задачі про брехунів і правдолюбців. Перші два правила, **`unwrap-true`** і **`unwrap-false`**, отримують вміст висловлювання у припущенні, що персонаж, якому належить висловлювання, є правдолюбцем або брехуном, і на цій підставі формують об'єкт **`claim`**.

;; Вилучення вмісту висловлювання.

```
(defrule unwrap-true
     (world (tag ?N)  (scope truth))
      (statement (speaker ?X)  (claim $?Y)  (tag ?N))
=>
        (assert (claim (content T ?X) (reason ?N)
                       (scope truth)))
        (assert (claim (content $?Y) (reason ?M)
                       (scope truth))))
(defrule unwrap-false
        (world (tag ?N)  (scope falsity))
         (statement (speaker ?X)  (claim $?Y)  (tag ?
N))
=>
        (assert (claim (content F  ?X)  (reason ?N)
               (scope falsity)))
        (assert (claim (content NOT $?Y)  (reason ?N)
                   (scope falsity)))
```

У кожному з наведених правил перший оператор в умовній частині робить припущення відповідно про правдивість чи брехливість персонажа, а другий оператор у заключній частині правила поширює припущення на твердження, що формуються – об'єкти **`claim`**.

Далее нам понадобятся правила, которые введут отрицания в выражения. Поскольку **`-T(A)`** эквивалентно **`F(A)`**, а **`-F(A)`** эквивалентно **`T(A)`**, то правила, выполняющие соответствующие преобразования, написать довольно просто. Анализ результатов применения этих правил значительно упростит выявление противоречий, следующих из определенного предположения.

```
;; Правила заперечення
(defrule not1
        ?F <- (claim (content NOT T ?P))
=>
        (modify ?F (content F ?P))
)
(defrule not2
        ?F <- (claim (conteny NOT F ?P))
=>
        (modify ?F (content T ?P))
)
```

```
;;; Виявлення протиріччя між припущенням про
;;; правдивості та наступними з нього фактами.
(defrule contra-truth
         (declare (salience 10))
         ?W <- (world (tag ?N) (scope truth))
         ?S  <- (statement (speaker ?Y) (tag ?N))
            ?P   <- (claim (content T ?X) (reason ?N)
(scope truth))
            ?Q <- (claim (content F ?X)   (reason ?N)
(scope truth))
=>
         (printout t crlf
                   "Statement is inconsistent if " ?Y "
is a knight."
```

```
;;; "Висловлювання суперечливо, якщо "? Y " правдолюб."
      t crlf)
      (retract ?Q)
      (retract ?P)
      (modify ?W (scope falsity))
)
```

Якщо припустити, що вихідне висловлювання було правдивим, то надалі виявляється суперечлива пара тверджень, які потім видаляються з робочої пам'яті, а значення правдивості припущення в об'єкті **`world`** змінюється на **`falsity`** (брехливість). Якщо ж після цього буде виявлено протиріччя, ми приходимо висновку про глобальної несумісності умов завдання, тобто. у цій постановці ми маємо справу з логічним парадоксом.

```
;;; Виявлення протиріччя між припущеннями про
;;; брехливості та наступними з нього фактами.
(defrule contra-falsity
        (declare (salience 10))
        ?W <- (world (tag ?N) (scope falsity))
        ?S   <- (statement (speaker ?Y) (tag ?N))
           ?P    <- (claim (content F ?X) (reason ?N)
(scope falsity))
        ?Q  <- (claim (content T ?X) (reason ?N) (scope
falsity))
=>
 (printout t crlf
  "Statement is inconsistent if " ?Y " is a knave."
;; "Висловлювання суперечливе, якщо "? Y " брехун."
     t crlf)
     (modify ?W (scope contra))
)
```

Правило **sweep** забезпечує перевірку, чи всі наслідки з неправильного припущення видалені з пам'яті.

```
;;; Видалити з бази фактів усі твердження,
;;; які випливають із припущення про правдивість.
(defrule sweep
      (declare (salience 20))
      (world (tag ?N) (scope falsity))
      ?F <- (claim (reason ?N) (scope truth))
=>
      (retract ?F)
)
```

Зверніть увагу, що правила **`contra-truth`**, **`contra-falsity`** і **`sweep`** мають вищий пріоритет (значення параметра **`salience`**), ніж інші правила. Цим забезпечується якомога раніше виявлення протиріччя, а отже, і видалення з бази фактів тверджень, зроблених на основі припущення, що призвело до протиріччя.

Якщо тепер запустити виконання програму, представивши їй вихідний набір фактів, відповідних умові завдання **`P0`**, то програма виявить, що обидва контексту суперечливі. Іншими словами, незалежно від того, чи припускаємо ми, що **`A`** говорить правду чи бреше, програма виявить суперечність у контексті **`world`**. Трасування програми у разі представлена у лістингу **`A.1`**. Рядки, виведені курсивом, – повідомлення основної програми, а інші – повідомлення програми трасування. Для зручності рядки, що вказують на активізацію правил, представлені жирним шрифтом.

Листинг А.1. Трассировка решения задачи P0

```
CLIPS> (reset)
= => f-0  (initial-fact)
= => f-1  (world (tag 1) (scope truth))
= => f-2  (statement (speaker A) (claim F A) (reason 0) (tag
1))
CLIPS> (run)
FIRE 1 unwrap-true: f-1,f-2
Assumption:
          A is a knight, so (T A) is true.
= => f-3  (claim (content F A) (reason 1) (scope truth))
= => f-4  (claim (content T A) (reason 1) (scope truth))
FIRE 2 contra-truth: f-1, f-2, f-4, f-3
Statement is inconsistent if A is a knight.
<= = f-3  (claim (content F A) (reason 1) (scope truth))
<= = f-4  (claim (content T A) (reason 1) (scope truth))
<= = f-1  (world (tag 1) (scope truth))
= => f-5  (world (tag 1) (scope falsity))
FIRE 3 unwrap-false: f-5, f-2
Assumption
A is a knave, so (T A) is false.
```

```
= => f-6 (claim (content NOT F A) (reason 1) (scope
falsity))
= => f-7 (claim (content F A) (reason 1) (scope falsity))
FIRE 4 not2: f-6
<= = f-6 (claim (content NOT F A) (reason 1) (scope
falsity))
= => f-8 (claim (content T A) (reason 1) (scope falsity))
FIRE 5 contra-falsity: f-5, f-2, f-7, f-8
Statement is inconsistent if A is a knave.
<= = f-5 (world (tag 1) (scope falsity))
= => f-9 (world (tag 1) (scope contra))
```

Наступна лекція буде присвячена мові програмування Triton.