МЕТОДИ І СИСТЕМИ ШТУЧНОГО ІНТЕЛЕКТУ

3 курс, весна 2021

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

Лекція 16

Приклади задач штучного інтелекту на CLIPS

Пропонується студентам ознайомитися з приведеними текстами програм реалізацій задач штучного інтелекту в системі CLIPS та запустити їх.

1. Експертна системна підбору вина до обіду

Ця експертна система вибирає відповідне вино для споживання під час їжі.

```
;;;======================================================
;;;   Wine Expert Sample Problem
;;;
;;;     WINEX: The WINe EXpert system.
;;;     This example selects an appropriate wine
;;;     to drink with a meal.
;;;
;;;     CLIPS Version 6.0 Example
;;;
;;;     To execute, merely load, reset and run.
;;;======================================================

(defmodule MAIN (export ?ALL))

;;****************
;;* DEFFUNCTIONS *
;;****************
```

```
(deffunction MAIN::ask-question (?question ?allowed-values)
   (printout t ?question)
   (bind ?answer (read))
   (if (lexemep ?answer) then (bind ?answer (lowcase ?answer)))
   (while (not (member ?answer ?allowed-values)) do
      (printout t ?question)
      (bind ?answer (read))
      (if (lexemep ?answer) then (bind ?answer (lowcase ?answer))))
   ?answer)

;;*****************
;;* INITIAL STATE *
;;*****************

(deftemplate MAIN::attribute
   (slot name)
   (slot value)
   (slot certainty (default 100.0)))

(defrule MAIN::start
  (declare (salience 10000))
  =>
  (set-fact-duplication TRUE)
  (focus QUESTIONS CHOOSE-QUALITIES WINES PRINT-RESULTS))

(defrule MAIN::combine-certainties ""
```

```
  (declare (salience 100)
           (auto-focus TRUE))
  ?rem1 <- (attribute (name ?rel) (value ?val) (certainty ?per1))
  ?rem2 <- (attribute (name ?rel) (value ?val) (certainty ?per2))
  (test (neq ?rem1 ?rem2))
  =>
  (retract ?rem1)
  (modify ?rem2 (certainty (/ (- (* 100 (+ ?per1 ?per2)) (* ?per1 ?per2))
100))))

;;******************
;;* QUESTION RULES *
;;******************

(defmodule QUESTIONS (import MAIN ?ALL) (export ?ALL))

(deftemplate QUESTIONS::question
   (slot attribute (default ?NONE))
   (slot the-question (default ?NONE))
   (multislot valid-answers (default ?NONE))
   (slot already-asked (default FALSE))
   (multislot precursors (default ?DERIVE)))

(defrule QUESTIONS::ask-a-question
   ?f <- (question (already-asked FALSE)
                   (precursors)
```

```
                    (the-question ?the-question)
                    (attribute ?the-attribute)
                    (valid-answers $?valid-answers))
   =>
   (modify ?f (already-asked TRUE))
   (assert (attribute (name ?the-attribute)
                      (value (ask-question ?the-question ?valid-answers)))))

(defrule QUESTIONS::precursor-is-satisfied
   ?f <- (question (already-asked FALSE)
                   (precursors ?name is ?value $?rest))
         (attribute (name ?name) (value ?value))
   =>
   (if (eq (nth 1 ?rest) and)
    then (modify ?f (precursors (rest$ ?rest)))
    else (modify ?f (precursors ?rest))))

(defrule QUESTIONS::precursor-is-not-satisfied
   ?f <- (question (already-asked FALSE)
                   (precursors ?name is-not ?value $?rest))
         (attribute (name ?name) (value ~?value))
   =>
   (if (eq (nth 1 ?rest) and)
    then (modify ?f (precursors (rest$ ?rest)))
    else (modify ?f (precursors ?rest))))
```

```
;;********************
;;* WINEX QUESTIONS *
;;********************

(defmodule WINE-QUESTIONS (import QUESTIONS ?ALL))

(deffacts WINE-QUESTIONS::question-attributes
  (question (attribute main-component)
            (the-question "Is the main component of the meal meat, fish, or
poultry? ")
            (valid-answers meat fish poultry unknown))
  (question (attribute has-turkey)
            (precursors main-component is poultry)
            (the-question "Does the meal have turkey in it? ")
            (valid-answers yes no unknown))
  (question (attribute has-sauce)
            (the-question "Does the meal have a sauce on it? ")
            (valid-answers yes no unknown))
  (question (attribute sauce)
            (precursors has-sauce is yes)
            (the-question "Is the sauce for the meal spicy, sweet, cream, or
tomato? ")
            (valid-answers sauce spicy sweet cream tomato unknown))
  (question (attribute tastiness)
            (the-question "Is the flavor of the meal delicate, average, or
strong? ")
```

```
            (valid-answers delicate average strong unknown))
  (question (attribute preferred-body)
            (the-question "Do you generally prefer light, medium, or full bodied
wines? ")
            (valid-answers light medium full unknown))
  (question (attribute preferred-color)
            (the-question "Do you generally prefer red or white wines? ")
            (valid-answers red white unknown))
  (question (attribute preferred-sweetness)
            (the-question "Do you generally prefer dry, medium, or sweet wines?
")
            (valid-answers dry medium sweet unknown)))

;;******************
;; The RULES module
;;******************

(defmodule RULES (import MAIN ?ALL) (export ?ALL))

(deftemplate RULES::rule
  (slot certainty (default 100.0))
  (multislot if)
  (multislot then))

(defrule RULES::throw-away-ands-in-antecedent
  ?f <- (rule (if and $?rest))
```

```
  =>
  (modify ?f (if ?rest)))

(defrule RULES::throw-away-ands-in-consequent
  ?f <- (rule (then and $?rest))
  =>
  (modify ?f (then ?rest)))

(defrule RULES::remove-is-condition-when-satisfied
  ?f <- (rule (certainty ?c1)
              (if ?attribute is ?value $?rest))
  (attribute (name ?attribute)
             (value ?value)
             (certainty ?c2))
  =>
  (modify ?f (certainty (min ?c1 ?c2)) (if ?rest)))

(defrule RULES::remove-is-not-condition-when-satisfied
  ?f <- (rule (certainty ?c1)
              (if ?attribute is-not ?value $?rest))
  (attribute (name ?attribute) (value ~?value) (certainty ?c2))
  =>
  (modify ?f (certainty (min ?c1 ?c2)) (if ?rest)))

(defrule RULES::perform-rule-consequent-with-certainty
  ?f <- (rule (certainty ?c1)
```

```
              (if)
              (then ?attribute is ?value with certainty ?c2 $?rest))
  =>
  (modify ?f (then ?rest))
  (assert (attribute (name ?attribute)
                     (value ?value)
                     (certainty (/ (* ?c1 ?c2) 100)))))

(defrule RULES::perform-rule-consequent-without-certainty
  ?f <- (rule (certainty ?c1)
              (if)
              (then ?attribute is ?value $?rest))
  (test (or (eq (length$ ?rest) 0)
            (neq (nth 1 ?rest) with)))
  =>
  (modify ?f (then ?rest))
  (assert (attribute (name ?attribute) (value ?value) (certainty ?c1))))

;;*******************************
;;* CHOOSE WINE QUALITIES RULES *
;;*******************************

(defmodule CHOOSE-QUALITIES (import RULES ?ALL)
                            (import QUESTIONS ?ALL)
                            (import MAIN ?ALL))
```

```
(defrule CHOOSE-QUALITIES::startit => (focus RULES))

(deffacts the-wine-rules

  ; Rules for picking the best body

  (rule (if has-sauce is yes and
            sauce is spicy)
        (then best-body is full))

  (rule (if tastiness is delicate)
        (then best-body is light))

  (rule (if tastiness is average)
        (then best-body is light with certainty 30 and
              best-body is medium with certainty 60 and
              best-body is full with certainty 30))

  (rule (if tastiness is strong)
        (then best-body is medium with certainty 40 and
              best-body is full with certainty 80))

  (rule (if has-sauce is yes and
            sauce is cream)
        (then best-body is medium with certainty 40 and
              best-body is full with certainty 60))
```

```
(rule (if preferred-body is full)
      (then best-body is full with certainty 40))

(rule (if preferred-body is medium)
      (then best-body is medium with certainty 40))

(rule (if preferred-body is light)
      (then best-body is light with certainty 40))

(rule (if preferred-body is light and
         best-body is full)
      (then best-body is medium))

(rule (if preferred-body is full and
         best-body is light)
      (then best-body is medium))

(rule (if preferred-body is unknown)
      (then best-body is light with certainty 20 and
            best-body is medium with certainty 20 and
            best-body is full with certainty 20))

; Rules for picking the best color

(rule (if main-component is meat)
```

```
      (then best-color is red with certainty 90))

(rule (if main-component is poultry and
          has-turkey is no)
      (then best-color is white with certainty 90 and
            best-color is red with certainty 30))

(rule (if main-component is poultry and
          has-turkey is yes)
      (then best-color is red with certainty 80 and
            best-color is white with certainty 50))

(rule (if main-component is fish)
      (then best-color is white))

(rule (if main-component is-not fish and
          has-sauce is yes and
          sauce is tomato)
      (then best-color is red))

(rule (if has-sauce is yes and
          sauce is cream)
      (then best-color is white with certainty 40))

(rule (if preferred-color is red)
      (then best-color is red with certainty 40))
```

```
(rule (if preferred-color is white)
      (then best-color is white with certainty 40))

(rule (if preferred-color is unknown)
      (then best-color is red with certainty 20 and
           best-color is white with certainty 20))

; Rules for picking the best sweetness

(rule (if has-sauce is yes and
          sauce is sweet)
      (then best-sweetness is sweet with certainty 90 and
           best-sweetness is medium with certainty 40))

(rule (if preferred-sweetness is dry)
      (then best-sweetness is dry with certainty 40))

(rule (if preferred-sweetness is medium)
      (then best-sweetness is medium with certainty 40))

(rule (if preferred-sweetness is sweet)
      (then best-sweetness is sweet with certainty 40))

(rule (if best-sweetness is sweet and
          preferred-sweetness is dry)
```

```
        (then best-sweetness is medium))

  (rule (if best-sweetness is dry and
            preferred-sweetness is sweet)
        (then best-sweetness is medium))

  (rule (if preferred-sweetness is unknown)
        (then best-sweetness is dry with certainty 20 and
              best-sweetness is medium with certainty 20 and
              best-sweetness is sweet with certainty 20))

)

;;************************
;;* WINE SELECTION RULES *
;;************************

(defmodule WINES (import MAIN ?ALL))

(deffacts any-attributes
  (attribute (name best-color) (value any))
  (attribute (name best-body) (value any))
  (attribute (name best-sweetness) (value any)))

(deftemplate WINES::wine
  (slot name (default ?NONE))
```

```
  (multislot color (default any))
  (multislot body (default any))
  (multislot sweetness (default any)))

(deffacts WINES::the-wine-list
  (wine (name Gamay) (color red) (body medium) (sweetness medium sweet))
  (wine (name Chablis) (color white) (body light) (sweetness dry))
  (wine (name Sauvignon-Blanc) (color white) (body medium) (sweetness dry))
  (wine (name Chardonnay) (color white) (body medium full) (sweetness medium
dry))
  (wine (name Soave) (color white) (body light) (sweetness medium dry))
  (wine (name Riesling) (color white) (body light medium) (sweetness medium
sweet))
  (wine (name Geverztraminer) (color white) (body full))
  (wine (name Chenin-Blanc) (color white) (body light) (sweetness medium sweet))
  (wine (name Valpolicella) (color red) (body light))
  (wine (name Cabernet-Sauvignon) (color red) (sweetness dry medium))
  (wine (name Zinfandel) (color red) (sweetness dry medium))
  (wine (name Pinot-Noir) (color red) (body medium) (sweetness medium))
  (wine (name Burgundy) (color red) (body full))
  (wine (name Zinfandel) (color red) (sweetness dry medium)))

(defrule WINES::generate-wines
  (wine (name ?name)
        (color $? ?c $?)
        (body $? ?b $?)
```

```
        (sweetness $? ?s $?))
  (attribute (name best-color) (value ?c) (certainty ?certainty-1))
  (attribute (name best-body) (value ?b) (certainty ?certainty-2))
  (attribute (name best-sweetness) (value ?s) (certainty ?certainty-3))
  =>
  (assert (attribute (name wine) (value ?name)
                     (certainty (min ?certainty-1 ?certainty-2 ?certainty-3)))))

;;*****************************
;;* PRINT SELECTED WINE RULES *
;;*****************************

(defmodule PRINT-RESULTS (import MAIN ?ALL))

(defrule PRINT-RESULTS::header ""
   (declare (salience 10))
   =>
   (printout t t)
   (printout t "        SELECTED WINES" t t)
   (printout t " WINE                    CERTAINTY" t)
   (printout t " ------------------------------" t)
   (assert (phase print-wines)))

(defrule PRINT-RESULTS::print-wine ""
  ?rem <- (attribute (name wine) (value ?name) (certainty ?per))
  (not (attribute (name wine) (certainty ?per1&:(> ?per1 ?per))))
```

```
  =>
  (retract ?rem)
  (format t " %-24s %2d%%%n" ?name ?per))

(defrule PRINT-RESULTS::remove-poor-wine-choices ""
  ?rem <- (attribute (name wine) (certainty ?per&:(< ?per 20)))
  =>
  (retract ?rem))

(defrule PRINT-RESULTS::end-spaces ""
   (not (attribute (name wine)))
   =>
   (printout t t))
```

2. Мавпи та банани

Це розширена версія досить поширеної проблеми планування ШІ. Справа в тому, щоб мавпа знайшла і з’їла трохи бананів **.**

Проблема мавпи і банана — проблема у галузі штучного інтелекту, особливо в логічному програмуванні та плануванні, яка відноситься до задач *пошуку в просторі станів*. Вона часто використовується для ілюстрації задач моделювання поведінки об'єктів. Уперше задача була запропонована Дж. Маккарті в 1963 році.

Постановка задачі: Мавпа перебуває в кімнаті, в якій підвішена до стелі купа бананів, недосяжна для мавпи. Однак у кімнаті є також стілець і палиця. Стеля має потрібну висоту, щоб мавпа, яка стояла на стільці, могла збити палицею банани. Мавпа вміє пересуватися, носити з собою інші речі, тягнутися до бананів і махати палицею в повітрі. Яка найкраща послідовність дій для мавпи?

Розв'язання проблеми: Мавпи мають здатність не тільки пам'ятати, як полювати і збирати, але й учитися нових речей, як це відбувається з мавпою та бананами: незважаючи на те, що мавпа, можливо, ніколи не потрапляла в ідентичну ситуацію. Мавпа здатна зробити висновок, що їй потрібно зробити сходи, розташувати їх нижче бананів і піднятися, щоб дотягнутися до них.

```
;;;======================================================
;;;   Monkees and Bananas Sample Problem
;;;
;;;     This is an extended version of a
;;;     rather common AI planning problem.
;;;     The point is for the monkee to find
;;;     and eat some bananas.
;;;
;;;     CLIPS Version 6.0 Example
;;;
;;;     To execute, merely load, reset and run.
;;;======================================================

;;;*************
;;;* TEMPLATES *
;;;*************
```

```
(deftemplate monkey
   (slot location
      (type SYMBOL)
      (default green-couch))
   (slot on-top-of
      (type SYMBOL)
      (default floor))
   (slot holding
      (type SYMBOL)
      (default blank)))

(deftemplate thing
   (slot name
      (type SYMBOL)
      (default ?NONE))
   (slot location
      (type SYMBOL)
      (default ?NONE))
   (slot on-top-of
      (type SYMBOL)
      (default floor))
   (slot weight
      (type SYMBOL)
      (allowed-symbols light heavy)
      (default light)))
```

```
(deftemplate chest
   (slot name
      (type SYMBOL)
      (default ?NONE))
   (slot contents
      (type SYMBOL)
      (default ?NONE))
   (slot unlocked-by
      (type SYMBOL)
      (default ?NONE)))

(deftemplate goal-is-to
   (slot action
      (type SYMBOL)
      (allowed-symbols hold unlock eat move on walk-to)
      (default ?NONE))
   (multislot arguments
      (type SYMBOL)
      (default ?NONE)))

;;;*************************
;;;* CHEST UNLOCKING RULES *
;;;*************************

(defrule hold-chest-to-put-on-floor ""
```

```
  (goal-is-to (action unlock) (arguments ?chest))
  (thing (name ?chest) (on-top-of ~floor) (weight light))
  (monkey (holding ~?chest))
  (not (goal-is-to (action hold) (arguments ?chest)))
  =>
  (assert (goal-is-to (action hold) (arguments ?chest))))

(defrule put-chest-on-floor ""
  (goal-is-to (action unlock) (arguments ?chest))
  ?monkey <- (monkey (location ?place) (on-top-of ?on) (holding ?chest))
  ?thing <- (thing (name ?chest))
  =>
  (printout t "Monkey throws the " ?chest " off the "
              ?on " onto the floor." crlf)
  (modify ?monkey (holding blank))
  (modify ?thing (location ?place) (on-top-of floor)))

(defrule get-key-to-unlock ""
  (goal-is-to (action unlock) (arguments ?obj))
  (thing (name ?obj) (on-top-of floor))
  (chest (name ?obj) (unlocked-by ?key))
  (monkey (holding ~?key))
  (not (goal-is-to (action hold) (arguments ?key)))
  =>
  (assert (goal-is-to (action hold) (arguments ?key))))
```

```
(defrule move-to-chest-with-key ""
  (goal-is-to (action unlock) (arguments ?chest))
  (monkey (location ?mplace) (holding ?key))
  (thing (name ?chest) (location ?cplace&~?mplace) (on-top-of floor))
  (chest (name ?chest) (unlocked-by ?key))
  (not (goal-is-to (action walk-to) (arguments ?cplace)))
  =>
  (assert (goal-is-to (action walk-to) (arguments ?cplace))))

(defrule unlock-chest-with-key ""
  ?goal <- (goal-is-to (action unlock) (arguments ?name))
  ?chest <- (chest (name ?name) (contents ?contents) (unlocked-by ?key))
  (thing (name ?name) (location ?place) (on-top-of ?on))
  (monkey (location ?place) (on-top-of ?on) (holding ?key))
  =>
  (printout t "Monkey opens the " ?name " with the " ?key
              " revealing the " ?contents "." crlf)
  (modify ?chest (contents nothing))
  (assert (thing (name ?contents) (location ?place) (on-top-of ?name)))
  (retract ?goal))

;;;**********************
;;;* HOLD OBJECT RULES *
;;;**********************

(defrule unlock-chest-to-hold-object ""
```

```
  (goal-is-to (action hold) (arguments ?obj))
  (chest (name ?chest) (contents ?obj))
  (not (goal-is-to (action unlock) (arguments ?chest)))
  =>
  (assert (goal-is-to (action unlock) (arguments ?chest))))

(defrule use-ladder-to-hold ""
  (goal-is-to (action hold) (arguments ?obj))
  (thing (name ?obj) (location ?place) (on-top-of ceiling) (weight light))
  (not (thing (name ladder) (location ?place)))
  (not (goal-is-to (action move) (arguments ladder ?place)))
  =>
  (assert (goal-is-to (action move) (arguments ladder ?place))))

(defrule climb-ladder-to-hold ""
  (goal-is-to (action hold) (arguments ?obj))
  (thing (name ?obj) (location ?place) (on-top-of ceiling) (weight light))
  (thing (name ladder) (location ?place) (on-top-of floor))
  (monkey (on-top-of ~ladder))
  (not (goal-is-to (action on) (arguments ladder)))
  =>
  (assert (goal-is-to (action on) (arguments ladder))))

(defrule grab-object-from-ladder ""
  ?goal <- (goal-is-to (action hold) (arguments ?name))
  ?thing <- (thing (name ?name) (location ?place)
```

```
                    (on-top-of ceiling) (weight light))
  (thing (name ladder) (location ?place))
  ?monkey <- (monkey (location ?place) (on-top-of ladder) (holding blank))
  =>
  (printout t "Monkey grabs the " ?name "." crlf)
  (modify ?thing (location held) (on-top-of held))
  (modify ?monkey (holding ?name))
  (retract ?goal))

(defrule climb-to-hold ""
  (goal-is-to (action hold) (arguments ?obj))
  (thing (name ?obj) (location ?place) (on-top-of ?on&~ceiling) (weight light))
  (monkey (location ?place) (on-top-of ~?on))
  (not (goal-is-to (action on) (arguments ?on)))
  =>
  (assert (goal-is-to (action on) (arguments ?on))))

(defrule walk-to-hold ""
  (goal-is-to (action hold) (arguments ?obj))
  (thing (name ?obj) (location ?place) (on-top-of ~ceiling) (weight light))
  (monkey (location ~?place))
  (not (goal-is-to (action walk-to) (arguments ?place)))
  =>
  (assert (goal-is-to (action walk-to) (arguments ?place))))

(defrule drop-to-hold ""
```

```
  (goal-is-to (action hold) (arguments ?obj))
  (thing (name ?obj) (location ?place) (on-top-of ?on) (weight light))
  (monkey (location ?place) (on-top-of ?on) (holding ~blank))
  (not (goal-is-to (action hold) (arguments blank)))
  =>
  (assert (goal-is-to (action hold) (arguments blank))))

(defrule grab-object ""
  ?goal <- (goal-is-to (action hold) (arguments ?name))
  ?thing <- (thing (name ?name) (location ?place)
                   (on-top-of ?on) (weight light))
  ?monkey <- (monkey (location ?place) (on-top-of ?on) (holding blank))
  =>
  (printout t "Monkey grabs the " ?name "." crlf)
  (modify ?thing (location held) (on-top-of held))
  (modify ?monkey (holding ?name))
  (retract ?goal))

(defrule drop-object ""
  ?goal <- (goal-is-to (action hold) (arguments blank))
  ?monkey <- (monkey (location ?place)
                   (on-top-of ?on)
                   (holding ?name&~blank))
  ?thing <- (thing (name ?name))
  =>
  (printout t "Monkey drops the " ?name "." crlf)
```

```
  (modify ?monkey (holding blank))
  (modify ?thing (location ?place) (on-top-of ?on))
  (retract ?goal))

;;;**********************
;;;* MOVE OBJECT RULES *
;;;**********************

(defrule unlock-chest-to-move-object ""
  (goal-is-to (action move) (arguments ?obj ?))
  (chest (name ?chest) (contents ?obj))
  (not (goal-is-to (action unlock) (arguments ?chest)))
  =>
  (assert (goal-is-to (action unlock) (arguments ?chest))))

(defrule hold-object-to-move ""
  (goal-is-to (action move) (arguments ?obj ?place))
  (thing (name ?obj) (location ~?place) (weight light))
  (monkey (holding ~?obj))
  (not (goal-is-to (action hold) (arguments ?obj)))
  =>
  (assert (goal-is-to (action hold) (arguments ?obj))))

(defrule move-object-to-place ""
  (goal-is-to (action move) (arguments ?obj ?place))
  (monkey (location ~?place) (holding ?obj))
```

```
  (not (goal-is-to (action walk-to) (arguments ?place)))
  =>
  (assert (goal-is-to (action walk-to) (arguments ?place))))

(defrule drop-object-once-moved ""
  ?goal <- (goal-is-to (action move) (arguments ?name ?place))
  ?monkey <- (monkey (location ?place) (holding ?obj))
  ?thing <- (thing (name ?name) (weight light))
  =>
  (printout t "Monkey drops the " ?name "." crlf)
  (modify ?monkey (holding blank))
  (modify ?thing (location ?place) (on-top-of floor))
  (retract ?goal))

(defrule already-moved-object ""
  ?goal <- (goal-is-to (action move) (arguments ?obj ?place))
  (thing (name ?obj) (location ?place))
  =>
  (retract ?goal))

;;;************************
;;;* WALK TO PLACE RULES *
;;;************************

(defrule already-at-place ""
  ?goal <- (goal-is-to (action walk-to) (arguments ?place))
```

```
  (monkey (location ?place))
  =>
  (retract ?goal))

(defrule get-on-floor-to-walk ""
  (goal-is-to (action walk-to) (arguments ?place))
  (monkey (location ~?place) (on-top-of ~floor))
  (not (goal-is-to (action on) (arguments floor)))
  =>
  (assert (goal-is-to (action on) (arguments floor))))

(defrule walk-holding-nothing ""
  ?goal <- (goal-is-to (action walk-to) (arguments ?place))
  ?monkey <- (monkey (location ~?place) (on-top-of floor) (holding blank))
  =>
  (printout t "Monkey walks to " ?place "." crlf)
  (modify ?monkey (location ?place))
  (retract ?goal))

(defrule walk-holding-object ""
  ?goal <- (goal-is-to (action walk-to) (arguments ?place))
  ?monkey <- (monkey (location ~?place) (on-top-of floor) (holding ?obj&~blank))
  =>
  (printout t "Monkey walks to " ?place " holding the " ?obj "." crlf)
  (modify ?monkey (location ?place))
  (retract ?goal))
```

```
;;;************************
;;;* GET ON OBJECT RULES *
;;;************************

(defrule jump-onto-floor ""
  ?goal <- (goal-is-to (action on) (arguments floor))
  ?monkey <- (monkey (on-top-of ?on&~floor))
  =>
  (printout t "Monkey jumps off the " ?on " onto the floor." crlf)
  (modify ?monkey (on-top-of floor))
  (retract ?goal))

(defrule walk-to-place-to-climb ""
  (goal-is-to (action on) (arguments ?obj))
  (thing (name ?obj) (location ?place))
  (monkey (location ~?place))
  (not (goal-is-to (action walk-to) (arguments ?place)))
  =>
  (assert (goal-is-to (action walk-to) (arguments ?place))))

(defrule drop-to-climb ""
  (goal-is-to (action on) (arguments ?obj))
  (thing (name ?obj) (location ?place))
  (monkey (location ?place) (holding ~blank))
  (not (goal-is-to (action hold) (arguments blank)))
```

```
  =>
  (assert (goal-is-to (action hold) (arguments blank))))

(defrule climb-indirectly ""
  (goal-is-to (action on) (arguments ?obj))
  (thing (name ?obj) (location ?place) (on-top-of ?on))
  (monkey (location ?place) (on-top-of ~?on&~?obj) (holding blank))
  (not (goal-is-to (action on) (arguments ?on)))
  =>
  (assert (goal-is-to (action on) (arguments ?on))))

(defrule climb-directly ""
  ?goal <- (goal-is-to (action on) (arguments ?obj))
  (thing (name ?obj) (location ?place) (on-top-of ?on))
  ?monkey <- (monkey (location ?place) (on-top-of ?on) (holding blank))
  =>
  (printout t "Monkey climbs onto the " ?obj "." crlf)
  (modify ?monkey (on-top-of ?obj))
  (retract ?goal))

(defrule already-on-object ""
  ?goal <- (goal-is-to (action on) (arguments ?obj))
  (monkey (on-top-of ?obj))
  =>
  (retract ?goal))
```

```
;;;*********************
;;;* EAT OBJECT RULES *
;;;*********************

(defrule hold-to-eat ""
  (goal-is-to (action eat) (arguments ?obj))
  (monkey (holding ~?obj))
  (not (goal-is-to (action hold) (arguments ?obj)))
  =>
  (assert (goal-is-to (action hold) (arguments ?obj))))

(defrule satisfy-hunger ""
  ?goal <- (goal-is-to (action eat) (arguments ?name))
  ?monkey <- (monkey (holding ?name))
  ?thing <- (thing (name ?name))
  =>
  (printout t "Monkey eats the " ?name "." crlf)
  (modify ?monkey (holding blank))
  (retract ?goal ?thing))

;;;***********************
;;;* INITIAL STATE RULE *
;;;***********************

(defrule startup ""
  =>
```

```
(assert (monkey (location t5-7) (on-top-of green-couch) (holding blank)))
(assert (thing (name green-couch) (location t5-7) (weight heavy)))
(assert (thing (name red-couch) (location t2-2) (weight heavy)))
(assert (thing (name big-pillow) (location t2-2) (on-top-of red-couch)))
(assert (thing (name red-chest) (location t2-2) (on-top-of big-pillow)))
(assert (chest (name red-chest) (contents ladder) (unlocked-by red-key)))
(assert (thing (name blue-chest) (location t7-7) (on-top-of ceiling)))
(assert (chest (name blue-chest) (contents bananas) (unlocked-by blue-key)))
(assert (thing (name blue-couch) (location t8-8) (weight heavy)))
(assert (thing (name green-chest) (location t8-8) (on-top-of ceiling)))
(assert (chest (name green-chest) (contents blue-key) (unlocked-by red-key)))
(assert (thing (name red-key) (location t1-3)))
(assert (goal-is-to (action eat) (arguments bananas))))
```

3. Канібали та месіонери

Задача про місіонерів і канібалів, і тісно пов'язана з ними задача про ревнивого чоловіка, класичні приклади «задач про перетин річки». Ця задача — іграшкова проблема в галузі штучного інтелекту, де вона була використана Саулом Амарелем як приклад проблеми подання.

У *місіонерів і канібалів* є проблема: троє місіонерів і троє канібалів повинні перетнути річку за допомогою човна, що здатен нести не більше двох осіб, при обмеженні, що на березі не може залишатися більше канібалів, ніж місіонерів (якщо вони б були, канібали з'їли б місіонерів.) Також човни не можуть перетнути річку по собі, без людей на борту.

Задача ревнивого чоловіка. Замість місіонерів і канібалів є троє одружених пар, з обмеженням, що жодна жінка не може бути в присутності іншої людини, якщо її чоловік також присутній. Відповідно до цього обмеження, не може бути чоловіки і жінки присутні на березі з переважаючими жінок чоловіками, тому що якщо б вони були, деякі жінки були б покинутими. Таким чином, при зміні чоловіків людожерами і жінок місіонерами, будь-яке рішення проблеми ревнивого чоловіка також стане рішення задачі місіонерів і канібалів.

В нашому прикладі класична проблема ШІ “канібали та місіонери” пропонується у сільськогосподарському плані. Сенс у тому, щоб перевезти фермера, лисицю (іноді вовка), капусту та козу через річку. Але човен вміщує лише 2 місця. Якщо лишитися наодинці з козою, лисиця її з’їсть. Якщо залишитися наодинці з капустою, коза її з’їсть.
Цей приклад використовує правила та збіг зразків фактів для вирішення проблеми.

```
;;;======================================================
;;;   Farmer's Dilemma Problem
;;;
;;;     Another classic AI problem (cannibals and the
;;;     missionary) in agricultural terms. The point is
;;;     to get the farmer, the fox the cabbage and the
;;;     goat across a stream.
;;;        But the boat only holds 2 items. If left
;;;     alone with the goat, the fox will eat it. If
;;;     left alone with the cabbage, the goat will eat
;;;     it.
;;;        This example uses rules and fact pattern
;;;     matching to solve the problem.
;;;
;;;     CLIPS Version 6.0 Example
;;;
;;;     To execute, merely load, reset and run.
;;;======================================================
```

```
(defmodule MAIN
  (export deftemplate status))

;;;*************
;;;* TEMPLATES *
;;;*************

;;; The status facts hold the state
;;; information of the search tree.

(deftemplate MAIN::status
   (slot search-depth (type INTEGER) (range 1 ?VARIABLE))
   (slot parent (type FACT-ADDRESS SYMBOL) (allowed-symbols no-parent))
   (slot farmer-location
      (type SYMBOL) (allowed-symbols shore-1 shore-2))
   (slot fox-location
      (type SYMBOL) (allowed-symbols shore-1 shore-2))
   (slot goat-location
      (type SYMBOL) (allowed-symbols shore-1 shore-2))
   (slot cabbage-location
      (type SYMBOL) (allowed-symbols shore-1 shore-2))
   (slot last-move
      (type SYMBOL) (allowed-symbols no-move alone fox goat cabbage)))

;;;*****************
```

```
;;;* INITIAL STATE *
;;;*****************

(deffacts MAIN::initial-positions
  (status (search-depth 1)
          (parent no-parent)
          (farmer-location shore-1)
          (fox-location shore-1)
          (goat-location shore-1)
          (cabbage-location shore-1)
          (last-move no-move)))

(deffacts MAIN::opposites
  (opposite-of shore-1 shore-2)
  (opposite-of shore-2 shore-1))

;;;************************
;;;* GENERATE PATH RULES *
;;;************************

(defrule MAIN::move-alone
  ?node <- (status (search-depth ?num)
                   (farmer-location ?fs))
  (opposite-of ?fs ?ns)
  =>
  (duplicate ?node (search-depth =(+ 1 ?num))
```

```
                    (parent ?node)
                    (farmer-location ?ns)
                    (last-move alone)))

(defrule MAIN::move-with-fox
  ?node <- (status (search-depth ?num)
                   (farmer-location ?fs)
                   (fox-location ?fs))
  (opposite-of ?fs ?ns)
  =>
  (duplicate ?node (search-depth =(+ 1 ?num))
                   (parent ?node)
                   (farmer-location ?ns)
                   (fox-location ?ns)
                   (last-move fox)))

(defrule MAIN::move-with-goat
  ?node <- (status (search-depth ?num)
                   (farmer-location ?fs)
                   (goat-location ?fs))
  (opposite-of ?fs ?ns)
  =>
  (duplicate ?node (search-depth =(+ 1 ?num))
                   (parent ?node)
                   (farmer-location ?ns)
                   (goat-location ?ns)
```

```
                  (last-move goat)))

(defrule MAIN::move-with-cabbage
  ?node <- (status (search-depth ?num)
                   (farmer-location ?fs)
                   (cabbage-location ?fs))
  (opposite-of ?fs ?ns)
  =>
  (duplicate ?node (search-depth =(+ 1 ?num))
                   (parent ?node)
                   (farmer-location ?ns)
                   (cabbage-location ?ns)
                   (last-move cabbage)))

;;;******************************
;;;* CONSTRAINT VIOLATION RULES *
;;;******************************

(defmodule CONSTRAINTS
  (import MAIN deftemplate status))

(defrule CONSTRAINTS::fox-eats-goat
  (declare (auto-focus TRUE))
  ?node <- (status (farmer-location ?s1)
                   (fox-location ?s2&~?s1)
                   (goat-location ?s2))
```

```
  =>
  (retract ?node))

(defrule CONSTRAINTS::goat-eats-cabbage
  (declare (auto-focus TRUE))
  ?node <- (status (farmer-location ?s1)
                   (goat-location ?s2&~?s1)
                   (cabbage-location ?s2))
  =>
  (retract ?node))

(defrule CONSTRAINTS::circular-path
  (declare (auto-focus TRUE))
  (status (search-depth ?sd1)
          (farmer-location ?fs)
          (fox-location ?xs)
          (goat-location ?gs)
          (cabbage-location ?cs))
  ?node <- (status (search-depth ?sd2&:(< ?sd1 ?sd2))
                   (farmer-location ?fs)
                   (fox-location ?xs)
                   (goat-location ?gs)
                   (cabbage-location ?cs))
  =>
  (retract ?node))
```

```
;;;**********************************
;;;* FIND AND PRINT SOLUTION RULES *
;;;**********************************

(defmodule SOLUTION
  (import MAIN deftemplate status))

(deftemplate SOLUTION::moves
   (slot id (type FACT-ADDRESS SYMBOL) (allowed-symbols no-parent))
   (multislot moves-list
      (type SYMBOL) (allowed-symbols no-move alone fox goat cabbage)))

(defrule SOLUTION::recognize-solution
  (declare (auto-focus TRUE))
  ?node <- (status (parent ?parent)
                   (farmer-location shore-2)
                   (fox-location shore-2)
                   (goat-location shore-2)
                   (cabbage-location shore-2)
                   (last-move ?move))
  =>
  (retract ?node)
  (assert (moves (id ?parent) (moves-list ?move))))

(defrule SOLUTION::further-solution
  ?node <- (status (parent ?parent)
```

```
                    (last-move ?move))
  ?mv <- (moves (id ?node) (moves-list $?rest))
  =>
  (modify ?mv (id ?parent) (moves-list ?move ?rest)))

(defrule SOLUTION::print-solution
  ?mv <- (moves (id no-parent) (moves-list no-move $?m))
  =>
  (retract ?mv)
  (printout t crlf "Solution found: " crlf crlf)
  (bind ?length (length ?m))
  (bind ?i 1)
  (bind ?shore shore-2)
  (while (<= ?i ?length)
     (bind ?thing (nth ?i ?m))
     (if (eq ?thing alone)
        then (printout t "Farmer moves alone to " ?shore "." crlf)
        else (printout t "Farmer moves with " ?thing " to " ?shore "." crlf))
     (if (eq ?shore shore-1)
        then (bind ?shore shore-2)
        else (bind ?shore shore-1))
     (bind ?i (+ 1 ?i))))
```

Очевидним узагальненням є зміна числа ревнивих пар (або місіонерів і канібалів), місткість судна, або обох параметрів. Якщо човен вміщує 2 осіб, то 2 пари вимагають 5 поїздок, від 4 або більше пар, завдання не має рішення.

Якщо човен може вмістити 3 чоловік, то може перевезти до 5 пар, якщо човен може містити 4-х осіб, то можливо перевезти будь-яку кількість пар.

Якщо в середині річки додати острів, то потім будь-яке число пар може перетнути річку за допомогою двох чоловік.

У наступній лекції ми ознайомимся зі складним прикладом створення експертної системи в CLIPS.