Università degli Studi di Bari
Dipartimento di Informatica

LACAM
Machine Learning

CLIPS: (Advanced) Pattern Matching & Functional Abstraction

Antonio Vergari

May 1, 2016

(Advanced) Pattern Matching

Variables and Wildcards

As already seen we can use ? and $? to represent variables for single or multifields in rules. They are called *wildcards* in CLIPS.

```
(defrule move-to-floor
    ?goal <- (goal (move ?block1) (on-top-of floor))
    ?stack-1 <- (stack ?block1 $?rest)
    =>
    (retract ?goal ?stack-1)
    (assert (stack ?block1))
    (assert (stack $?rest))
    (printout t ?block1 " moved on top of floor." crlf))
```

When it is not necessary to bind variable names across multiple patters or functions, one can use anonymous wildcards:

```
(defrule is-parent ;; family_dag_template.clp
    "asserting parenthood"
    (person (name ?name) (gender ?) (children $?))
    =>
    (assert (is-parent (name ?name))) ;; no need for $? or ?
```

Field Constraints I

Rules as advanced queries can specify constraints to the referred field values in a LHS pattern.
Imagine this deftemplate to conceptualize a bit of knowledge about films:

```
(deftemplate movie
    (slot title (type STRING) (default ?NONE))
    (slot genre (type SYMBOL)
        (allowed-symbols noir romance action cartoon indie surreal))
    (slot main-actor (type STRING))
    (multislot actors (type STRING))
    (slot director (type STRING))
    (slot producer (type STRING)))
```

To express a query about film not belonging to a genre we can use the operator ~ (*not*):

```
(defrule interesting-movies
    "Defining interesting films as those not belonging to the romance genre"
    (movie (title ?title) (genre ~romance))
    =>
    (printout t "An interesting movie " ?title crlf))
```

Field Constraints II

Similarly to the not constraint, one can express an or constraing, allowing a subset of values to make a field matchable. They can be separated by a pining |:

```
(defrule hipster—movies
    (movie (title ?title) (genre indie | surreal))
    =>
    (printout t "Probably an hipster movie " ?title crlf))
```

The alternative was to write two different rules:

```
(defrule hipster—movies—I
    (movie (title ?title) (genre indie))
    =>
    (printout t "Probably an hipster movie " ?title crlf))

(defrule hipster—movies—II
    (movie (title ?title) (genre surreal))
    =>
    (printout t "Probably an hipster movie " ?title crlf))
```

Field Constraints III

The and field constraing, expressed with an ampersand **&**, allows for an arbitrary concatenation of field constrains on a field pattern:

```
(defrule intellectual—movies
    (movie (title ?title) (genre ?genre&~romance&~action))
    =>
    (printout t "Likely a movie for intellectuals: ", ?title crlf))
```

The and constraint can be used to express constraints involving other fields in different pattern values as well:

```
(defrule uncle—fc ;; back to the family DAG example
    (parent ?grandparent ?father)
    ;; what happens if we do not put this constraint?
    (parent ?grandparent ?uncle&~?father)
    (parent ?father ?child)
    (male ?uncle)
    =>
    (assert (uncle—fc ?uncle ?child)))
```

Field Constraints IV

Predicate field constraint

In conjunction with the and operator **&** one can specify a custom constraing in the form of a function predicate on the referred variable with **:**,

```
(defrule eddie—murphy—movies
    "Listing films with Eddie Murphy as actor"
    (movie (title ?title)
           ;; member$ tests if an elem is in a multifield
           (actors $?actors&:(member$ "Eddie Murphy" $?actors)))
    =>
    (printout t "A film with Eddie Murphy: " ?title crlf))
```

To get the titles of movies engrossing more than a threshold:

```
(defrule blockbuster—movies
    (movie (title ?title) (box—office ?total&:(> ?total 500.0)))
    =>
    (printout t "Surely a blockbuster: " ?title crlf))
```

Test predicate

To introduce constraints in the LHS in a similar fashion we can employ the **test** predicate function. It evaluates the truth of a condition expressed by operators like **eq**, **neq**, **>**, **<**,...Here it is another take on the rule for derive the uncle relationship:

```
(defrule uncle
    (parent ?grandparent ?father) (parent ?grandparent ?uncle)
    (parent ?father ?child) (male ?uncle)
    (test (neq ?uncle ?father)) ;; a test predicate pattern
    =>
    (assert (uncle ?child2 ?child)))
```

How to retrieve movies where the director is also the main actor and producer:

```
(defrule homemade-movies
    (movie (title ?title)
    (director ?director) (producer ?producer) (main-actor ?actor))
    (test (eq ?producer ?director ?actor))
    =>
    (printout t "This is a one made film: " ?title crlf))
```

And/Or Conditional Elements

CLIPS automatically considers all the patterns in a LHS to be in a logical AND. An **or** CE specifies two or more patterns in disjunction. As for the or field constraint, it is a shortcut to write more compact rules.

```
(defrule hipster—movies—or
    (or (movie (title ?title) (genre indie))
        (movie (title ?title) (genre surreal)))
    =>
    (printout t "Probably an hipster movie (OR): " ?title crlf))
```

Not Conditional Element

It is possible to express a pattern as the absence of a fact in the WM via the **not** CE.
To match the names of orphans in the family DAG one can write:

```
(defrule orphan—2
     (or (male ?person)
         (female ?person))
     (or (not (father ?parent ?person))
         (not (mother ?parent ?person))
         (and
              (not (father ?parent ?person))
              (not (mother ?parent ?person))))
     =>
     (assert (orphan—2 ?person)))
```

Be prepared to unexpected behaviors with the *closed-world assumption* (if something is not asserted in the WM is assumed to be false, e.g. if you do not generate some father and mother facts for all possible parenthood relationships, all persons will be considered orphans).

Exists Conditional Element

Similarly the **exists** CE checks for *at least one fact in the WM* matching the referred pattern.

The difference with a simple pattern is that it generates only one rule activation.

```
(defrule one-blockbuster
    "Check if there has been a movie surpassing 500"
    (exists (movie (box-office ?total&:(> ?total 500.0))))
    =>
    (printout t "There was one movie to engross > 500.0" crlf))
```

It is implemented with a combination of and and not CE:

```
(defrule one-blockbuster-I
    "Check if there has been a movie surpassing 500"
    (and (initial-fact)
        (not (not (movie (box-office ?total&:(> ?total 500.0))))))
        =>
        (printout t "There was one movie to engross > 500.0" crlf))
```

Functional Abstraction and Programming

The Game of Life

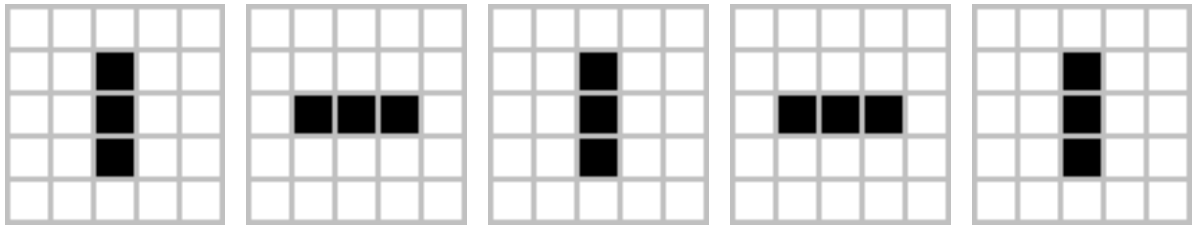
To explore a little bit of functional abstraction and programming in CLIPS we will use **Conway's Game of Life (GoL)**. It is a formal system (*cellular automaton*) in the form of a 2D finite grid in which cells can be alive (active) or dead (empty).
The rules governing the evolution of the system are these two:

I. a live cell with exactly 2 or 3 live cells as neighbors lives, otherwise dies

II. a dead cell with exactly 3 live cells as neighbors becomes alive

The neighbors of a cell are the 8 adjacent cells surrounding it.

An example of a sequence of world statuses in a 5×5 grid is:



A very simple conceptualization

If we want to implement GoL in CLIPS we shall represent the (current) *world status* and the *evolution rules*.

We will use an ordered facts of N^2 boolean values to represent a grid of $N \times N$ cells (what are the (dis)advantages?).
The representation for the previous example at time t_0 is:

```
(deffacts initial_world
    (world FALSE FALSE FALSE FALSE FALSE
           FALSE FALSE TRUE FALSE FALSE
           FALSE FALSE TRUE FALSE FALSE
           FALSE FALSE TRUE FALSE FALSE
           FALSE FALSE FALSE FALSE FALSE))
```

We will model the evolutionary rules *via user defined functions* this time, and use only one rule to update the current world status.

deffunction

CLIPS allows programmers to do functional abstraction and define their own functions with the **deffunction** construct:

```
(deffunction <function-name>
    ?<optional-comment>
    (*<regular-parameter> ?<multifield-parameter>)
    *<action>)
```

As in functional programming, the returned value is the last evaluated expression. Since we need to transform matrix coordinates into vectorial ones we could write:

```
(deffunction vec-coords
    "Converts a pair of matrix coordinates into vector ones"
    (?x ?y ?width)
    (+ (* (- ?x 1) ?width) ?y))
```

bind

To declare a variable with a local scope inside a function and initialize it to a value, one can use the **bind** function.

```
(bind ?x 5)     (bind ?x (** 2 5))
```

One could assign the value of an already defined variable as well:

```
(bind ?x (* ?y ?x)) ;; x *= y
```

Outside the scope in which bind is called the variable is undefined

```
CLIPS> (bind ?x (** 2 5))
32.0
CLIPS> (printout t ?x crlf) ;; CLIPS 6.26!!
[EVALUATN1] Variable x is unbound
CLIPS> (printout t ?x crlf) ;; CLIPS 6.30
32.0
CLIPS> (deffunction nth-pow-of-two (?n) (bind ?x (** 2 ?n)) ?x)
CLIPS> (printout t ?x crlf)
[EVALUATN1] Variable x is unbound
CLIPS> (printout t (nth-pow-of-two 5) crlf)
32.0
```

Multifield functions I

Multifields represent the simplest data structure one can operate on in CLIPS: heterogeneous lists.
Here are some useful functions on them: create$ initializes a list with the arguments provided. Passing no arguments results in the empty list.

```
CLIPS> (create$ hammer drill saw screw pliers wrench)
(hammer drill saw screw pliers wrench)
CLIPS> (create$ (+ 3 4) (* 2 3) (/ 8 4))
(7 6 2.0)
```

member$ returns a the first position of the first argument if it appears in the multifield passed as the second argument, FALSE otherwise:

```
CLIPS> (member$ "tarm" (create$ a disc—by "tarm"))
3
CLIPS> (member$ b (create$ a disc—by "tarm"))
FALSE
```

Multifield functions II

nth$ lets one access the element of a multifield (second argument) specified as the first argument. *Positions start from 1, not 0*.
In our conceptualization for GoL, to check if a cell is currently alive or dead we can simply check its vector position in the world status:

```
(deffunction is—cell—alive
    (?x ?y ?width $?world)
    (nth (vec—coords ?x ?y ?width) ?world))
```

first$ and rest$ are functions that return the sublists containing the first element of a multifield and all the remaining ones respectively:

```
1   CLIPS> (first$ (create$ block1 block2 block3))
2   (block1)
3   CLIPS> (rest$ (create$ block1 block2 block3))
4   (block2 block3)
```

(defglobal ?*num − rows* = 5)

Multifield functions III

length$ returns the number of elements in a multifield:

```
CLIPS> (length$ (create$ block1 block2 block3))
3
CLIPS> (deffunction assert—length (?list)
(eq (length$ ?list) (+ (length$ (first$ ?list)) (length$ (rest$ ?list)))))
CLIPS> (assert—length (create$ block1 block2 block3))
TRUE
```

insert$ and delete$ are the functions to modify a multifield as a list. delete$ needs two indices as 3rd and 4th arguments to indicate the slice to remove from the list. Remember that all objects are passed by value.

```
CLIPS> (insert$ (create$ block1 block2 block3) 1 block4)
(block4 block1 block2 block3)
CLIPS> (delete$ (create$ block1 block2 block3) 3 3)
(block1 block2)
CLIPS> (delete$ (create$ block1 block2 block3) 1 3)
()
```

String functions

A brief list of some useful system defined string functions:

```
CLIPS> (sym-cat "a string-" a-symbol) ;; symbol concatenation
a string-a-symbol
CLIPS> (str-cat "first" "-" "second") ;; string concatenation
"first-second"
CLIPS> (sub-string 3 4 "1234567") ;; substring extraction
"34"
CLIPS> (str-index "string" "This is a long string") ;; member
16
CLIPS> (str-index "string" "This is a long sentence")
FALSE
CLIPS> (eval "(+ 2 4 (* 5 10))") ;; evaluating CLIPS commands
56
CLIPS> (upcase "This is a lower case string") ;; to upper case
"THIS IS A LOWER CASE STRING"
CLIPS> (str-compare "aabba" "aaaba") ;; string comparison
1
CLIPS> (str-length "This string has 24 chars") ;; length
24
```

defglobal

Global variables can be defined, accessed and modified as well, with the construct defglobal:

```
(defglobal ?*num_rows* = 5)
(defglobal ?*num_cols* = 5)
```

To update a global variable one shall use the **bind** function:

```
(bind ?*num_rows* (+ ?*num_rows* 1))
```

Global variables can be used to define the only four values *you will ever need* to specify a rule salience:

```
(defglobal ?*highest_priority* = 1000)
(defglobal ?*high_priority* = 100)
(defglobal ?*normal_priority* = 10)
(defglobal ?*low_priority* = 1)
```

Each fact in the WM can be thought of as a global variable, however defglobals are useful since updating them does not activate rules.

Procedural constructs I

if-then-else

Being multiparadigmatic, CLIPS allows programming functions in a procedural way by introducing constructs for conditional statements and iterative loops.

Branching can be done with the **if…then…else** function, which takes a condition to be evaluated as a first argument and function calls on the two branches to be evaluated according to the initial condition returned value.

```
CLIPS> (if (> 100 2) then a else b)
a
```

A simple procedure to print a multifield by exploiting recursion:

```
(deffunction print-list
    "An utility function that prints the contents of a multifield"
    ($?list)
    (if (> (length$ ?list) 0)
        then (printout t (nth$ 1 ?list))
             (print-list (rest$ ?list)) ;; recursive call
        else (printout t crlf)))
```

Procedural constructs II

while loop

There are several constructs implementing loops in CLIPS. while needs a condition as first element and optionally a do keyword, then a sequence of expressions as function calls:

```
(while <expression> ?do
       <action>*)
```

A very short way to create a CLIPS meta-interpreter as an infinite loop:

```
(while (> 1 0) ;; always TRUE
    (printout t (eval (readline)) crlf))
```

The REPL is implemented with the functions readline, printout and eval which calls the interpreter on the input string representing a valid CLIPS command.

Procedural constructs III

for loop

To implement a loop based on the increment of a counter, one can employ the loop-for-count function. It takes as first argument a list representing the variable to use as a counter, its range and eventually increment.

Here is a procedure to print the current world status by looping through the cells of the grid as a matrix (two nested loops):

```
(deffunction print-world
    "Printing the world as a grid with . (dead cells) and X (alive ones)"
    (?height ?width $?world)
    (loop-for-count (?i 1 ?height)
        (loop-for-count (?j 1 ?width)
            (if (is-cell-alive ?i ?j ?width ?world)
                then (printout t " X ")
                else (printout t " . ")))
        (printout t crlf)))
```

Procedural constructs IV

return

One can explicitly use the **return** function to end one function call and passing the **return** function parameter as output.
This procedure calculates the list of legal neighbor cells given a cell and returns it as a multifield:

```
(deffunction get-8-neighborhood
    (?x ?y ?max-x ?max-y)
    (bind ?neighbors (create$))
    (loop-for-count (?i -1 1)
        (loop-for-count (?j -1 1)
            (if (and (>= (+ ?x ?i) 1)
                     (<= (+ ?x ?i) ?max-x)
                     (>= (+ ?y ?j) 1)
                     (<= (+ ?y ?j) ?max-y)
                     (or (<> ?i 0) (<> ?j 0)))
                then (bind ?neighbors
                     (create$ ?neighbors (+ ?x ?i) (+ ?y ?j)))))))
    return ?neighbors)
```

This return statement is superfluous.

GoL II

After computing the list of neighbor cells coordinates, we have to determine the number of alive cells among them:

```
(deffunction num-alive-neighbors
(?x ?y ?height ?width $?world)
(bind ?num-alive-neigh 0)
(bind ?neighborhood (get-8-neighborhood ?x ?y ?height ?width))
(bind ?length (div (length$ ?neighborhood) 2))
(loop-for-count (?i 1 ?length)
    (bind ?neigh-x (nth$ (- (* ?i 2) 1) ?neighborhood))
    (bind ?neigh-y (nth$ (* ?i 2) ?neighborhood))
    (if (is-cell-alive ?neigh-x ?neigh-y ?width ?world)
        then (bind ?num-alive-neigh (+ ?num-alive-neigh 1))))
return ?num-alive-neigh)
```

GoL III

Once we know the exact number of alive neighbor cells, we can determine whether or not a cell will live or die in the next world configuration:

```
(deffunction will-the-cell-live
    (?x ?y ?height ?width $?world)
    (bind ?is-alive (is-cell-alive ?x ?y ?width $?world))
    (bind ?n-alive (num-alive-neighbors ?x ?y ?height ?width $?world))
    (if ?is-alive
        then (if (or (< ?n-alive 2) (> ?n-alive 3))  ;; rule I
                 then (return FALSE)
                 else (return TRUE))
        else (if (= ?n-alive 3)                      ;; rule II
                 then (return TRUE)
                 else (return FALSE))))
```

GoL IV

Putting it all together, we can call a function that, in order to create a new world status, represents it as a multifield computed by updating the current status elements according to the will-the-cell-live function.

```
(deffunction evolve—world
    "Creating a new world status"
    (?height ?width $?world)
    (bind ?next—world (create$))
    (loop—for—count (?i 1 ?height)
        (loop—for—count (?j 1 ?width)
            (bind ?cell—status FALSE)
            (if (will—the—cell—live ?i ?j ?height ?width ?world)
                then (bind ?cell—status TRUE))
            (bind ?next—world (create$ ?next—world ?cell—status))))
    return ?next—world)
```

GoL V

The rule to evolve the world status will just retract the current configuration, compute the new one with evolve-world, assert and print it, update the global variable for the time and ask the user for input (as a possibility to exit the evol loop).

```
(defrule update-world
    "A simple rule to update the world configuration"
    ?world <- (world $?world-status)
    =>
    (retract ?world)
    (bind ?new-world-status (evolve-world ?*num-rows* ?*num-cols*
                                          ?world-status))
    (assert (world ?new-world-status))
    (bind ?*time* (+ ?*time* 1)) ;; updating time
    (printout t "New world (at time " ?*time* "): " t crlf)
    (print-world ?*num-rows* ?*num-cols* ?new-world-status)
    (assert (key-command (get-char t))) ;; press 'e' to exit
    (printout t crlf))
```

GoL VI

Like in the MU Game, we can halt the evolution by checking for a predefined key in a high priority rule:

```
(defrule end-evolution
    (declare (salience ?*high-priority*))
    ?k <- (key-command 101) ;; this can become a global const
    =>
    (retract ?k)
    (printout t crlf "Ending evolution" crlf)
    (halt))
```

A possible sequential output for the example:

```
New world (at t 1):     New world (at t 2):     New world (at t 3):
. . . . .               . . . . .               . . . . .
. . . . .               . . X . .               . . . . .
. X X X .               . . X . .               . X X X .
. . . . .               . . X . .               . . . . .
. . . . .               . . . . .               . . . . .
```

Exercises

more GoL

Write two functions to save to and load from a file a world configuration (in an arbitrary formalism).

Think about how to embed the store and retrieve functionalities into the current *rule cycle*. The user should be asked which action to take after each generation has been displayed:

proceed	to the next generation
save	the current world state to a user specified file
load	a world state (becoming the new current generation) from a user specified file
halt	the evolutionary process

For the primitive functions to `open`/`close`, `read`/`write`,...from a file refer to the manuals.

A simple diagnostic classifier I

In this exercise you have to implement a simple classifier that is able to diagnose a disease based on some observed symptoms.

Some symptoms will be already represented as already gained knowledge (facts in the WM), others shall be acquired from the user (a doctor that has checked a patient) through a question driven interaction.

The first step will be to design a conceptualization for this domain in the terms of facts.

The second step will be to design the rules to derive the diagnosis from the asserted facts, and the rules to assert facts from the question answers.

A simple diagnostic classifier II

These are the rules to reach a disease as a diagnosis.

I. **If** the skin and eyes are yellowish, **then** there is *icterus* (partial diagnosis)

II. **If** the eyes are yellowish but not the skin, **then** there is *scleral icterus* (partial diagnosis)

III. **If** scleral icterus is present, there is no fever, and the patient's stressed or without food, **then** *Gilbert's syndrome* can be diagnosed

IV. **If** icterus is present, there is fever, the patient is young and tired, dyspepsia has been diagnosed and the liver is enlarged, **then** *Acute viral hepatitis* can be diagnosed

V. **If** there is icterus, fever and the patient is not young but has recurrent pains and cholecyst pains, **then** it is *cholecystitis*

VI. **If** there is icterus, there is no fever and the patient is not young and abuses alchool, the liver is enlarged and the spleen is enlarged as well, **then** it is *alchoolic cirrhosis*

A simple diagnostic classifier III

For instance a symptom about the enlarged liver could be represented as a very simple ordered fact:

```
(symptom enlarged—liver TRUE)
```

but any conceptualization should be fine as long as you can match it in your LHSs.

Concerning rules, some ones shall have higher priority than others, for instance those asserting the diagnosed disease, so that the interaction can be halted when a diagnosis is present.

A simple diagnostic classifier IV

This is an example of a possible conversation:

```
*** A simple diagnostic classifier ***

Has the patient got a fever? (yes/y/no/n): y
Has the patient yellowish eyes? (yes/y/no/n): y
Has the patient yellowish skin? (yes/y/no/n): y
Is the patient stressed? (yes/y/no/n): n
Is the patient without food? (yes/y/no/n): n
Is the patient young? (yes/y/no/n): y
Is the patient tired? (yes/y/no/n): y
Has the patient been diagnosed dyspepsia? (yes/y/no/n): y
Has the patient's liver enlarged? (yes/y/no/n): y

>>>> Diagnosed disease: Acute viral hepatitis <<<<
```

A simple diagnostic classifier V

You can use these functions[1] to ask questions and collect answers:

```
(deffunction ask-question (?question $?allowed-values)
    (printout t ?question)
    (bind ?answer (read))
    (if (lexemep ?answer) ;; TRUE is ?answer is a STRING or SYMBOL
        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)

(deffunction yes-or-no-p (?question)
    (bind ?question (sym-cat ?question " (yes/y/no/n): "))
    (bind ?response (ask-question ?question yes no y n))
    (if (or (eq ?response yes) (eq ?response y))
        then TRUE
        else FALSE))
```

[1] Taken from Riley's and Giarratano's code.